# Intel® IXP1200 Network Processor Family

## Development Tools User's Guide

*December, 2001*

Order Number: 278302-009

# Revision History

| Revision Date | Revision | Description |
|---|---|---|
| 7/30/99 | .5 | Beta 1 release. |
| 8/30/99 | 001 | Beta 2 release. |
| 11/5/99 | 002 | Beta 3 release. |
| 3/3/00 | 003 | Beta 4 release. |
| 6/2/00 | 004 | Version 1.0 release. |
| 10/2/00 | 005 | Version 1.1 release. |
| 12/20/00 | 006 | Version 1.2 release. |
| 5/21/01 | 007c | Version 1.3 SDK release. (includes C Compiler) |
| 6/1/01 | 007 | Version 1.3 release. |
| 8/10/01 | 008 | Version 2.0 SDK |
| 12/5/01 | 009 | Version 2.01 SDK |

# Contents

# Figures

# Tables

# Introduction 1

## 1.1 About this Document

This manual is a reference for using the Intel® IXP1200 Network Processor development tools. The rest of this book is organized as follows:

Chapter 2, "Developer Workbench," describes the IXP1200 Workbench and its graphical user interface (GUI).

Chapter 3, "Assembler," describes how to run the IXP1200 Assembler.

Chapter 4, "Microengine C Compiler," describes how to run the IXP1200 Microengine C Compiler.

Chapter 5, "Linker," describes how to run the IXP1200 Linker.

Chapter 6, "Transactor," describes the IXP1200 Transactor and its commands.

Chapter 7, "Foreign Model Simulation Extensions," provides information on interfacing the IXP1200 to foreign models.

Appendix A, "Transactor States," contains a listing and descriptions of commonly used transactor states.

Appendix B, "Developer Workbench Shortcuts," contains a listing and descriptions of GUI shortcuts and buttons with references to Chapter 2 where they are used.

The intended audience for this book is Developers and Systems Programmers.

## 1.2 Related Documents

Further information on the IXP1200 is available in the following documents:

*IXP1200 Network Processor Family Microcode Programmer's Reference Manual*—Contains detailed programming information for designers.

*IXP1200 Network Processor Datasheet*—Contains summary information on the IXP1200 including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

*IXP1250 Network Processor Datasheet*—Contains summary information on the IXP1250 including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

*IXP1200 Network Processor Family Hardware Reference Manual* - Contains detailed hardware technical information for designers.

*IXP1200 Network Processor Microcode Software Reference Manual* - Contains detailed software technical information for designers.

*IXP1200 Microengine C Language Support Reference Manual* - Contains functions and intrinsics supported by the C Compiler for designers.

# Developer Workbench 2

## 2.1 Overview

The Intel® IXP1200 Network Processor Family Developer Workbench is an integrated development environment for assembling, linking, and debugging microcode that runs on the IXP1200 Network Processor Family Microengines. The Workbench is a Microsoft* Win32* application that runs on Windows NT* platforms.

### Features

Important Workbench features include:

- Source level debugging.
- Execution history and statistics.
- IX Bus device and network traffic simulation.
- Optional command line interface to the IXP1200 Network Processor Family Transactor.
- Command line interface to the IXP Network Processor simulators (Transactors).
- Command line interface to the IXP Network Processor simulators (Transactors).
- Customizable graphical user interface (GUI) components.

### Debugging Support

The Workbench supports debugging in four different configurations:

- **Local simulation with no foreign model**, in which the Workbench and the IXP1200 Network Processor Family simulator (Transactor) both run on the same Microsoft Windows* platform.
- **Local simulation with a local foreign model**, in which the Workbench, the Transactor, and a foreign model Dynamic-Link Library all run on the same Windows platform.
- **Local simulation with a remote foreign model**, in which the Workbench and the Transactor both run on the same Windows platform and communicate over the network with a foreign model running on a remote system.
- **Hardware**, in which the Workbench runs on a Windows host and communicates over a network or a serial port with a subsystem containing actual IXP12nn Network Processors.

### Performance Statistics

When debugging in a simulation configuration, the Workbench provides performance statistics for the IXP12nn Network Processor subsystem. For example, it provides data on memory bandwidth to the SDRAM. (See Section 2.11.15.)

## 2.2 Getting Help

You can get help about the Developer Workbench and the IXP1200 Network Processor Family several ways:

- On the **Help** menu, click **Help Topics**. This gives the Developer Workbench online help tool.
- On the **Help** menu, click **Tip of the Day**. The **Tip of the Day** dialog box displays helpful in formation about using the Workbench. The Tip of the Day changes every time you invoke it.
- In the **Project Workspace** window (see Figure 2-1), click the **InfoView** tab. This give you access to Workbench documentation. See Section 2.6.3, "About InfoView."."
- On the Web, go to http://developer.intel.com/design/network/products/npfamily/ixp1200.htm to get more information about Intel products.

### Developer Workbench Revision Information

To do this:

- On the **Help** menu, click **About Developer Workbench**.

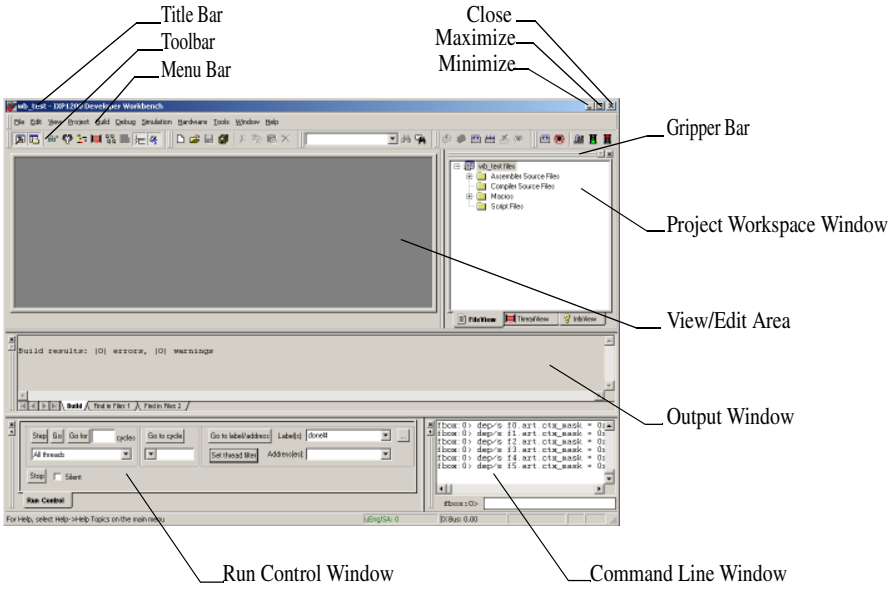
  The **About Developer Workbench** information box appears showing you the revision of your Developer Workbench.

## 2.3 About the Graphical User Interface (GUI)

The Workbench GUI (Figure 2-1) conforms to the standard Windows look and feel. You can do the following:

- **Dock and undock** (float) windows, menu bars, and toolbars (see Section 2.3.1).
- **Hide and show** windows and toolbars (see Section 2.3.2).
- **Customize** toolbars and menu bars (see Section 2.3.3).
- **Save and restore** GUI customizations (see Section 2.3.3.6).

**Figure 2-1. The Developer Workbench GUI**

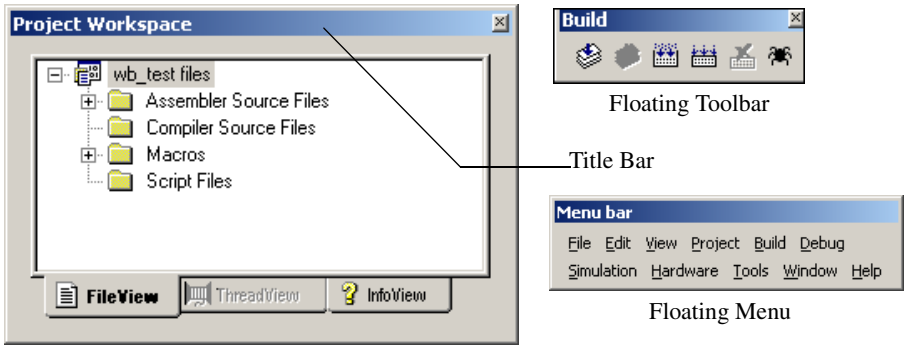



## 2.3.1 About Windows, Toolbars, and Menus

Dockable windows contain controls and data. You can attach them to a location on the Workbench main window or you can float them over the main window. All toolbars and menu bars are dockable. (See Figure 2-2.)

To float, or undock, a window or toolbar, double-click its gripper bar (see Figure 2-1). To restore it to its previously docked location, double-click its title bar. You can also drag the window to a new docking location.

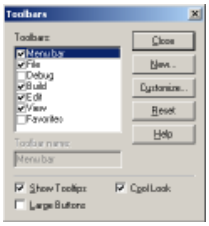
**Figure 2-2. Floating Window, Tool Bar, and Menu Bar**



## 2.3.2 Hiding and Showing Windows and Toolbars

Form the **View** menu, you can toggle the visibility of the following windows in the Workbench's GUI:

**Toolbar**



If you are viewing a source file, or the edit/view area is empty, selecting **Toolbar** on the **View** menu displays the **Toolbars** dialog box. Here you can select to view or clear to hide any of the available toolbars. You can also select **Show Tooltips, Large Buttons**, and **Cool Look**. (**Cool Look** removes the borders from the buttons.)

If you are viewing a thread, selecting or clearing **Toolbar** on the **View** menu shows or hides the toolbar at the top of the thread window.

**Workbook Mode**


This control puts the tabs at the bottom of the view/edit area (see **Figure 2-1**). Without the tabs you must use other methods to select different windows, such as going to the **Window** menu and selecting the window; cascading the windows using the button and selecting with the mouse pointer; using the Microengine and Thread lists (above, right). Removing the tabs gives you more workspace in the windows.

**Project Workspace** See **Section 2.6**.

**Output Window** This window displays the results of **Find in Files**, assembly and compile results, build results and other messages. See **Figure 2-1**.

Click the button to show or hide this window.

**Debug Windows**

| | |
|---|---|
| **Command Line** | See Section 2.11.5. |
| **Data Watch** | See Section 2.11.12. |
| **Memory Watch** | See Section 2.11.13. |
| **History** | See Section 2.11.16. |
| **Thread Status** | See Section 2.11.18. |
| **Queue Status** | See Section 2.11.17. |
| **IX Bus Device Status** | See Section 2.11.23.4. |
| **Run Control** | See Sections 2.11.8 and 2.11.9. |

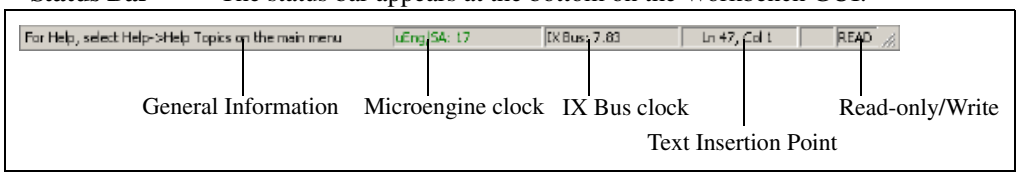
To toggle the visibility of a dockable window, select or clear the window's name on the **View** menu.

If a window is visible, you can hide it by clicking the button in either the upper-right or upper-left corner of the window.

If a toolbar is floating, you can hide it by clicking the button in the upper right corner.

*Note:* You can float and dock the GUI's default menu bar but you cannot hide it. If you create a customized menu bar, you can display or hide in it using the same method used for windows and toolbars.

**Status Bar** The status bar appears at the bottom on the Workbench GUI.




| | |
|---|---|
| **General Information** | Information and tips appear here as you work. |
| **Microengine Clock** | The present cycle count of the Microengine clock. |
| **IX Bus Clock** | The present cycle count of the IX Bus clock. |
| **Read-only/Write** | The Read/Write status of the selected file. If READ is dimmed, the status is Read/Write. |
| **Text Insertion Point** | The location of the text insertion point (cursor) by line and column. |

## 2.3.3 Customizing Toolbars and Menus

You can add and remove buttons from toolbars and create your own toolbars.

### 2.3.3.1 Creating Toolbars

To create a toolbar:

1. On the **Tools** menu, click **Customize**.

   The **Customize** dialog box appears.
2. Click the **Toolbars** tab.
3. Click **New**.

   The **New Toolbar** dialog box appears.
4. Type a name for the new toolbar and click **OK**.

The toolbar name is added to the **Toolbars** list and the new toolbar appears in a floating state. If you want the toolbar to be docked, drag it to the desired location.

To populate the toolbar with buttons, go to Section 2.3.3.4.

### 2.3.3.2 Renaming Toolbars

You can rename toolbars that you have created.

To rename a toolbar:

1. On the **Tools** menu, click **Customize**.

   The **Customize** dialog box appears.
2. Click the **Toolbars** tab.

3. Select the desired toolbar in the **Toolbars** list.
4. Edit the name in the **Toolbar Name** box at the bottom.
5. Click **OK**.

*Note:* You cannot rename the GUI's default toolbars (Menu bar, File, Debug, Build, Edit, View).

### 2.3.3.3 Deleting Toolbars

To delete a toolbar you have created:

1. On the **Tools** menu, click **Customize**.

   The **Customize** dialog box appears.
2. Click the **Toolbars** tab.
3. Select the toolbar to delete in the **Toolbars** list.
4. Click **Delete**.

*Note:* You cannot delete the GUI's default toolbars (Menu bar, File, Debug, Build, Edit, View).

### 2.3.3.4 Adding and Removing Toolbar Buttons and Controls

To customize the buttons on the toolbars:

1. On the **Tools** menu, click **Customize**.

   The **Customize** dialog box appears.
2. Click the **Commands** tab.
3. From the **Categories** list, select a command category.

   A set of toolbar buttons for that category appears in the **Buttons** area.

   To get a description of the command associated with a button, click the button. The description appears in the **Description** area at the bottom of the dialog box.
4. To place a button in a toolbar, drag the button to a location on a toolbar.
5. To remove a button from a toolbar, drag the button into the dialog box.
6. Click **OK** when done.

### 2.3.3.5 Customizing Menus

You can change the appearance of the main menu or you can put menus on toolbars.

#### Main Menu Appearance

To change the order of the main menu items:

1. On the **Tools** menu, click **Customize**.

   The **Customize** dialog box appears.
2. Drag any main menu item to the new position on the main menu bar. For example, drag **File** and drop it after **Help**.

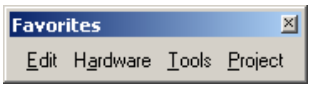
3. To remove a menu from the main menu bar, drag it into the work area below.
4. To add a menu to the main menu bar:
   a. In the **Customize** dialog box, click the **Commands** tab.
   b. Click **Menu** in the **Commands** box.
      All the menus appear in the **Buttons** box.
   c. Select a menu and drag it to the main menu bar.
      That menu then becomes a new menu on the main menu bar.

#### Menus On Toolbars

To put a menu on a toolbar:

1. In the **Customize** dialog box, click the **Commands** tab.
2. Click **Menu** in the **Categories** box.
   All the menus appear in the **Buttons** box.
3. Drag any menu to any toolbar.

*Note:* You can put your most used or favorite menus on a floating toolbar by creating a new toolbar (see example at right) and dragging the menus to that toolbar.




### 2.3.3.6 Returning to Default Toolbar Settings

To set toolbars to their default configurations:

1. On the **Tools** menu, click **Customize**.
   The **Customize** dialog box appears.
2. Click the **Toolbars** tab.
3. Select the desired toolbar and click **Reset.**

Only the Workbench default toolbars can be reset.

## 2.3.4 GUI Toolbar Configurations

#### Build Versus Debug

The Workbench maintains two sets of toolbar and docking configurations, one for debug mode and one for build, or non-debug mode. The GUI configuration that you establish while in build mode applies only when you are in build mode. Similarly, the debug mode GUI configuration applies only for debug mode.

#### Save And Restore

Menu bar and toolbar configurations are saved when you exit the Workbench. These configurations persist from one Workbench session to the next.

## 2.4 About Projects

A project consists of one or more IXP1200 Network Processor Family chips, microcode source files, debug script files, and Assembler, Compiler, and Linker settings used to build the microcode image files. This project configuration information is maintained in a developer workbench project file (.dwp).

You can:

| | |
|---|---|
| **Create** a new project. | See Section 2.4.1. |
| **Open** an existing project. | See Section 2.4.2. |
| **Save** a project. | See Section 2.4.3. |
| **Close** a project. | See Section 2.4.4. |

### 2.4.1 Creating a New Project

To create a new project:

1. On the **File** menu, click **New Project**.

   The **New Project** dialog box appears.

2. Type the name of the new project in the **Project name** box.
3. Specify a folder where you want to store the project in the **Location** box.

   If the folder doesn't exist, the Workbench creates it. You can browse to select the folder by clicking the ... button.

4. Specify the number of IXP1200 Network Processor Family chips to be in the project in the **Specify the chips to be in the project** box.
   - If you have only one chip in your project, it can be <unnamed>.
   - To specify more than one chip, they must all have unique names. You cannot add a second chip until you have named the first chip. When you finish creating a project, you cannot change the number of chips in it.
     - — To rename a chip, select the chip in the list and click **Rename**. The **Chip Name** dialog box appears. Type the chip's name and click **OK**.
     - — To add a chip to the configuration, click **Add**. The **Chip Name** dialog box appears. Type the chip's name and click **OK**.
     - — To delete a chip from the configuration, select the chip in the list and click **Delete**.
5. In the **Specify a chip** area, elect the type of IXP12nn processor you want simulated. You can change this setting at any time as part of the Chip Configuration (see Section 2.5.1).
6. When you are finished, click **OK** to create the project.

The project name you typed actually becomes a folder containing two files—project_name.dwp and project_name.dwo. From this point on, all the project files and information defaults to this folder or one of its subfolders. For example, a project named CrossBar has a project file named Crossbar.dwp.

*Note:* Creating a new project automatically closes the active project, if one is open, and asks you if you want to save any changes if there are any.

## 2.4.2 Opening a Project

To open an existing project:

1. On the **File** menu, click **Open Project**.

   The **Open Project** dialog box appears.

2. Browse to the folder that contains the project file (*.dwp) for the project you want to open.
3. Double-click the project filename or select the project filename and click **Open**.

You can also select a project from the most recently used list of projects, if it is one of the most recent four projects opened.

1. On the **File** menu, click **Recent Projects**.
2. Click the project file from the list.

*Note:* Opening a project automatically closes the currently open project, if any, after asking you if you want to save changes if there are any.

## 2.4.3 Saving a Project

To save a modified project:

- On the **File** menu, click **Save Project**.

This saves all project configuration information to the project file. If your project hasn't been modified, the **Save Project** selection is unavailable. Also, on the **File** menu, click **Save All** to save all files and the current project.

The project is saved in the folder that you specified when you created it. If you opened an existing project, it is saved in the folder from which that you opened it.

*Note:* You do not have the option of saving the project in a different folder.

## 2.4.4 Closing a Project

To close a project:

- On the **File** menu, click **Close Project**.

  If there are any modified but unsaved files in the opened project, you are asked if you want to save these changes.

  — Click **Yes** to save the file and close it, or

  — Click **No** to close it without saving any changes, or

  — Click **Cancel** to abort closing the project.

An open project is automatically closed whenever you open another project or create a new project.

## 2.5 Configuring the System

To change the system configuration you must have a project opened.

You can configure the following:

| | |
|---|---|
| **Chip** | See Section 2.5.1. |
| **Memory** | See Section 2.5.2. |
| **IX Bus Interface** | See Section 2.5.3. |
| **PCI Interface** | See Section 2.5.4. |
| **IX Bus** | See Section 2.5.5. |

The settings in these pages are passed to the Transactor when debugging is started. They are not used during hardware debugging.

### 2.5.1 Modifying the Chip Configuration

To modify your project's chip:

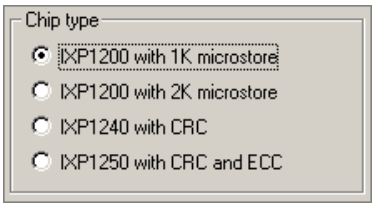
1. On the **Project** menu, click **System Configuration**.

   The **System Configuration** dialog box appears.

2. Click the **Chip** tab.

#### 2.5.1.1 Selecting Chip Type

Select the IXP12nn processor chip that you want to simulate. Note that you need to insure that the Assembler settings are compatible with your chip type selection (see Section 2.8.2).




The **IXP1200 with 1K microstore** is considered to be revision 1.

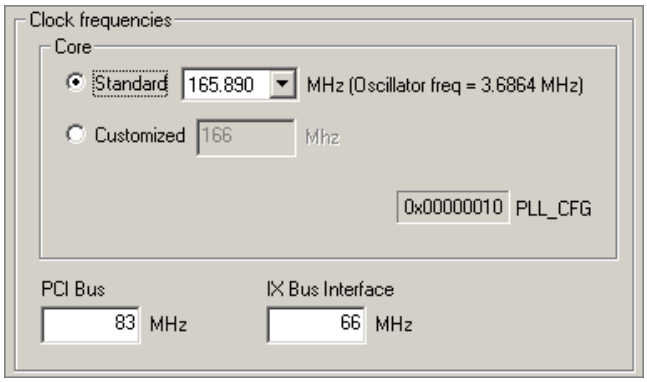
The **IXP1200 with 2K microstore** is considered to be revision 3.

The **IXP1240** is considered to be revision 6.

The **IXP1250** is considered to be revision 7.

### 2.5.1.2 Selecting Clock Frequencies

#### Standard Core




You can chose from one of the **Standard** core clock frequencies based on an oscillator frequency of 3.6864 MHz. Note that only the standard frequencies are supported in actual hardware. The **PLL_CFG** register data appears in the lower right corner. (See the *Programmer's Reference Manual* for more information.)

#### Customized Core

You can enter a customized clock frequency by clicking **Customized** and then typing a number from 0 to 1000. Note the **PLL_CFG** data in the lower right corner is unavailable as it no longer applies.

#### PCI Bus

You can change the **PCI Bus** clock frequency from the default of 83 to a value between 0-366 by selecting the number in the box and typing the new number.

#### IX Bus Interface

You can change the **IX Bus Interface** clock frequency from the default of 66 to a value between 0-366 by selecting the number in the box and typing the new number.

## 2.5.2 Modifying the Memory Configuration

To modify your chip's memory configuration:

1. On the **Project** menu, click **System Configuration**.

   The **System Configuration** dialog box appears.

2. Click the **Memory** tab.

If your project has multiple chips, select a chip from the **Select a chip** list. The settings on the page apply only to that chip.

***Note:*** You must select the number of chips when you created the project. (See Section 2.4.1.)

The settings controlled on this page and in the dialog boxes invoked from this page determine the values for the SDRAM and SRAM control and status registers. These register values are displayed in the **Registers** area on the right side of the page. (See the *Programmer's Reference Manual* for more information on these registers.)

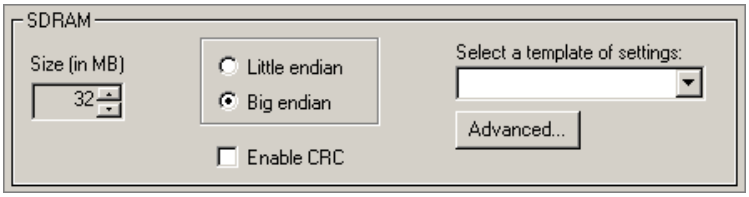
### 2.5.2.1 Changing the SDRAM Configuration

***Note:*** You cannot change the System Configuration if the Transactor is running (debug mode). If you start and stop the Transactor, you can no longer change the memory size.

The **SDRAM** area contains the controls for changing the SDRAM configuration.

To do this:

1. Specify memory size in the **Size** box.
2. Select whether the SDRAM unit treats data in big-endian (default) or little-endian format.




#### CRC

If you chose the IXP1240 or the IXP1250 on the **Chip** tab, you have the option of enabling CRC by selecting or clearing **Enable CRC.**

#### SDRAM Templates

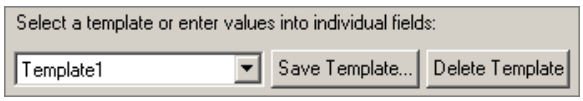
You can specify settings for the SDRAM registers indirectly by selecting a template from the **Select a template of settings** list.

To customize the SDRAM settings:

1. Click **Advanced**.

   The **SDRAM Configuration** dialog box appears.
2. Select a template from the list, or

   Specify individual fields in the registers.




To save your customized settings:

1. Click **Save Template**.

   The **Save Template** dialog box appears.
2. Type a unique name for the template.
3. Click **OK**.

To delete a template:

1. Select the template from the list in the **SDRAM Configuration** dialog box.
2. Click **Delete Template**.

***Note:*** Templates are independent of Workbench projects and are available across Workbench invocations.

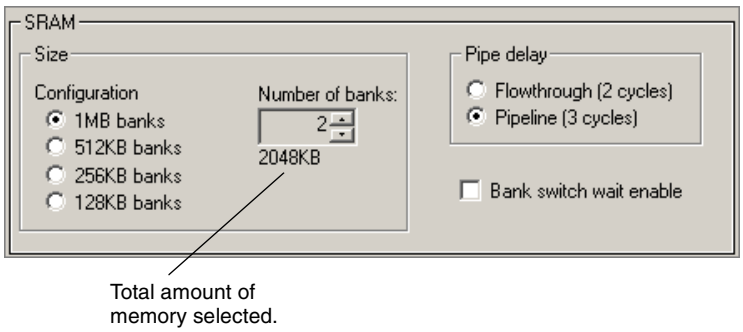
### 2.5.2.2 Changing the SRAM Configuration

***Note:*** You cannot change the System Configuration if the Transactor is running (debug mode). If you start and stop the Transactor, you can no longer change the memory size.

The **SRAM** area contains the controls for changing the SRAM configuration.

To do this:

1. Select which bank Configuration you want under **Configuration**.
2. Specify the number of banks (1-8) in the **Number of banks** box.
3. Select the type of pipe delay in the **Pipe delay** box.




4. Click **Bank switch wait enable** if desired. When enabled, this bit enables a wait state to be inserted when switching between the 8 SRAM banks. This bit must be set when using SRAMs that have a slower output disable time than output enable time.

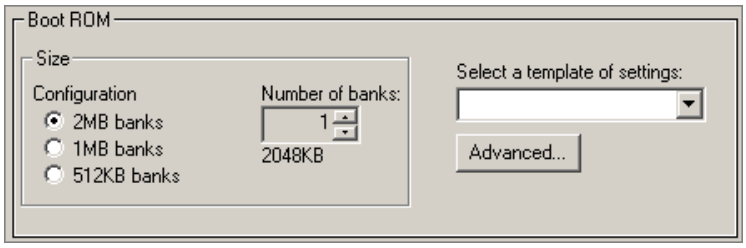
Note that the total size of SRAM displayed below the **Number of banks** box dynamically changes as you make your selections.

### 2.5.2.3 Changing the Boot ROM Configuration

***Note:*** You cannot change the System Configuration if the Transactor is running (debug mode). If you start and stop the Transactor, you can no longer change the memory size.

The **Boot ROM** area contains the controls for changing the Boot ROM configuration.

1. Select which bank configuration you want under **Configuration.**
2. Specify the number of banks (1-4) in the **Number of banks** box.




#### Boot ROM Templates

The settings for the SRAM registers that pertain to the Boot ROM can be specified indirectly by selecting a template from the **Select a template of settings** list.

To customize your settings:

1. Click **Advanced**.

   The **Boot ROM Configuration** dialog box appears.

2. Select a template from the list or specify the SRAM_SLOW_CONFIG and the SRAM_BOOT_CONFIG values that you want in the corresponding boxes.

To save your customized settings:

1. Click **Save Template**.

   The **Save Template** dialog box appears.

2. Type a unique name for the template.
3. Click **OK**.

To delete a template:

1. Select the template in the **Select a template...** list.
2. Click **Delete Template**.

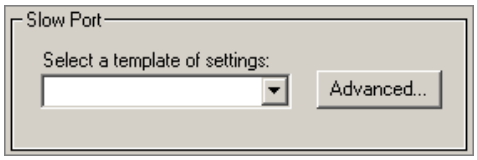
***Note:*** Templates are independent of Workbench projects and are available across Workbench invocations.

### 2.5.2.4 Changing the Slow Port Configuration

The **Slow Port** area contains the controls for changing the Slow Port configuration.

#### Slow Port Templates

The settings for the SRAM registers that pertain to Slow Port can be specified indirectly by selecting a template from the **Select a template of settings** list.




To customize your settings:

1. Click **Advanced**.

   The **Slow Port Configuration** dialog box appears.

2. Select a template from the list or you can specify the SRAM_SLOW_CONFIG and SRAM_SLOWPORT_CONFIG values that you want in the corresponding boxes.

To save your customized settings:

1. Click **Save Template**.

   The **Save Template** dialog box appears.

2. Specify a unique name for the template.
3. Click **OK**.

To delete a template:

1. Select the template.
2. Click **Delete Template**.

***Note:*** Templates are independent of Workbench projects and are available across Workbench invocations.

## 2.5.3 Modifying the IX Bus Interface Configuration

To modify your project's IX Bus Interface:

1. On the **Project** menu, click **System Configuration**.

   The **System Configuration** dialog box appears.

2. Click the **IX Bus Interface** tab.

If your project has multiple chips, select a chip from the **Select a chip** list. The settings on the page apply only to the chip you select.

The settings for the IX Bus Interface registers can be specified:

- Indirectly by selecting a template from the list, or
- Directly using the controls on the page.

### IX Bus Interface Templates

To save your customized settings:

1. Click **Save Template**.

   The **Save Template** dialog box appears.

2. Type a unique name for the template.
3. Click **OK**.

To delete a template:

1. Select the template from the **Select a template...** list.
2. Click **Delete Template**.

*Note:* Templates are independent of Workbench projects and are available across Workbench invocations.

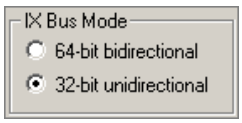
The settings controlled on this page and on the dialog boxes invoked from this page determine the values for the IXBI control and status registers. These register values are displayed in the **Registers** area in the lower right portion of the page.

### Timing Mode


To set the timing mode for the IX Bus and the Ready Bus:

- If you have 1 or 2 MACs/Devices, click the top selection.
- For 3 or more, click the bottom selection.

### IX Bus Mode




Select:

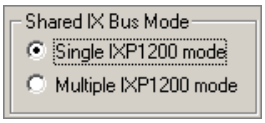
- 64-bit bidirectional, or
- 32-bit (dual) unidirectional.

### Status Mode


Select:

- MAC/Device returns a status (after EOP).
- MAC/Device doesn't return a status (after EOP).

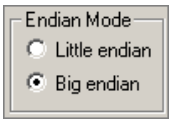
### Shared IX Bus Mode




Select:

- Single IXP1200 mode.
- Multiple IXP1200 mode (enable token passing).

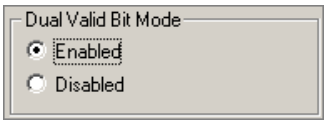
### Endian Mode




Controls how the IX Bus FBE# signals are interpreted when EOP is asserted during an IX Bus receive cycle. Default is Big endian.

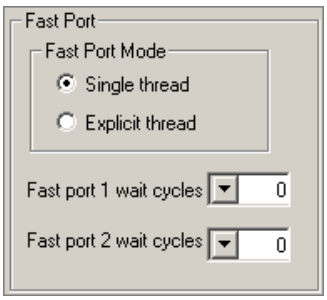
In the **Endian Mode** area, click **Little endian** or **Big endian** (default).

### Dual Valid Bit Mode



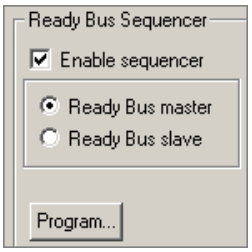

Selects the dual valid bit mode for TFIFO writes. Click **Enable** or **Disable**.

### Fast Port




- Click **Single thread** or **Explicit thread**. (See the *IXP1200 Programmers Reference Manual* for more details.)
- Select from 0-15 from the **Fast port 1 wait cycles** list. This is the number of IX Bus clock cycles for Fast Port 1.
- Select from 0-15 from the **Fast port 2 wait cycles** list. This is the number of IX Bus clock cycles for Fast Port 2.

## 2.5.3.1 Programming the Ready Bus Sequencer

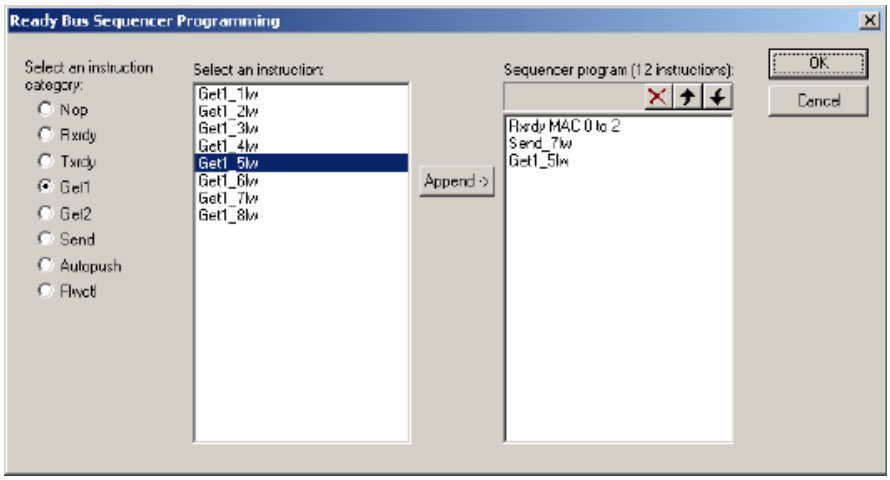



To program the Ready Bus Sequencer:

1. Select **Enable sequencer** in the **Ready Bus Sequencer** area.
2. Click **Program**.

   The **Ready Bus Sequencer Programming** dialog box appears.
3. Select the instruction category from the set of option buttons on the left side of the dialog box.
4. Select an instruction from the list of instructions in that category.

5. Click **Append** to insert that instruction at the end of the program.




6. Repeat steps 3-5 above until you have the desired 12 instructions in the **Sequencer Program** window.

   Since the program requires exactly 12 instructions, the **Append** button becomes unavailable once 12 instructions are in the program.

You can delete an instruction from the program by:

1. Selecting the instruction in the **Sequencer program** area.
2. Clicking the  button.

You can also move an instruction up or down in the program by:

1. Selecting the instruction in the **Sequencer program** area.
2. Clicking the up  or the down  arrow.

When you have completed the 12 instruction program, click **OK**.

### 2.5.3.2 Configuring Autopush

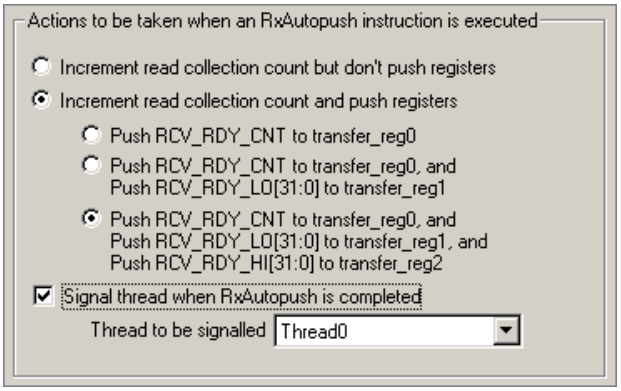
To specify what actions are to be performed when the sequencer executes an RxAutopush or TxAutopush instruction:

1. Click **Configure Autopush**.

   The **Autopush** dialog box appears.

2. Specify whether or not registers are to be pushed, and, if so, which ones.

3. Specify whether a specific thread is to be signalled then select the thread from the list.




There are two areas in the **Configure Autopush** dialog box, one for RxAutopush and one for TxAutopush. You can configure these actions independently.

## 2.5.4 Modifying the PCI Interface Configuration

To modify your project's PCI Interface Configuration:

1. On the **Project** menu, click **System Configuration**.

   The **System Configuration** dialog box appears.

2. Click the **PCI Interface** tab.

If your project has multiple chips, you must select a chip from the **Select a chip** list. The settings on the page apply only to that chip.

#### PCI Interface Configuration Templates

You can specify the settings for the PCI Interface registers indirectly by selecting a template from the list, or directly using the controls on the page.

To save your customized settings:

1. Click **Save Template**.

   The **Save Template** dialog box appears.

2. Type a unique name for the template.

To delete a template:

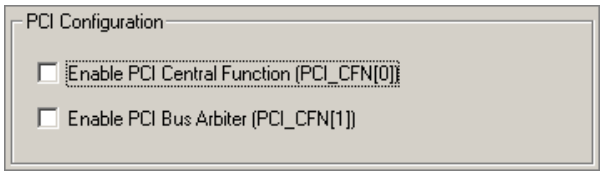
1. Select the template.
2. Click **Delete Template**.

*Note:* Templates are independent of Workbench projects and are available across Workbench invocations.

The settings controlled on this page and on the dialog boxes invoked from this page determine the values for the PCI Interface control and status registers. These register values are displayed in the **Registers** area in the lower right portion of the page.

## 2.5.4.1 About PCI Configuration

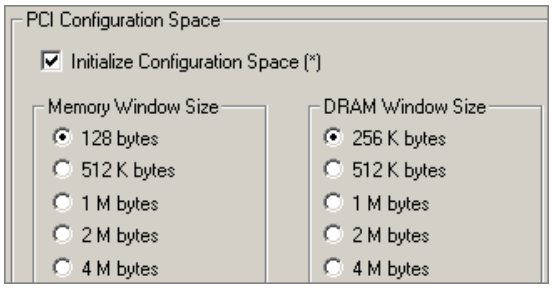
### PCI Bus Arbiter and Central Function




In the **PCI Configuration** area, specify whether you want the IXP12nn to act as the PCI central function, the PCI bus arbiter, or both. A system must have exactly one PCI bus arbiter. If you do not have a foreign model that is acting as the arbiter, you must enable one of your chips to act as the arbiter. Similarly, only one chip can be the PCI central function.

## 2.5.4.2 About PCI Configuration Space

### PCI Configuration Space




You can specify that the Workbench initialize the PCI configuration space by enabling **Initialize Configuration Space**. However, the initialization is not done if Intel® StrongARM* core model is enabled because it is assumed that the core programs will do the initialization.

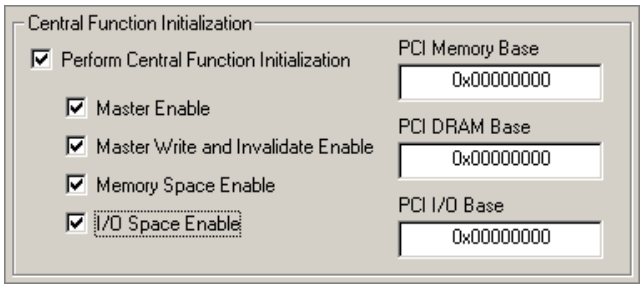
### Memory Window Size

You can specify the window sizes for the memory and DRAM spaces by clicking the window size in the **Memory Window Size** area and the **DRAM Window Size** area.

### I/O Space Window

The I/O space window size is fixed at 128 bytes.

### Central Function initialization




- To specify that the Workbench perform central function initialization, enable **Perform Central Function Initialization**.

  This enables the other selections in the group.
- If you check **Memory Space Enable**, you must specify a **PCI Memory Base** and a **PCI DRAM Base** in the corresponding boxes.
- If you check **I/O Space Enable**, you must specify a **PCI I/O Base** in the corresponding box.

*Note:* For debugging convenience, you can have the Workbench perform central function initialization even if you have not enabled the IXP12nn to be the PCI central function.

For more information on central function initialization, see the PCI_CMD_STAT register bits <4,2:0> in the *Programmer's Reference Manual.*

## 2.5.5 Modifying the IX Bus Configuration

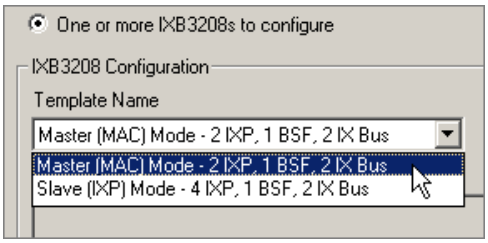
To modify your project's IX Bus Configuration:

1. On the **Project** menu, click **System Configuration**.

   The **System Configuration** dialog box appears.

2. Click the **IX Bus** tab.
3. If you are not using an IXB3208 chip in your configuration, click **No IXB3208 to configure**.

To customize the bus configuration for multichip projects, see Section 2.5.5.1.




If your project has two chips and you want a bus configuration that includes an IXB3208 chip configured in MAC mode:

1. Click **One or more IXB3208 to configure**.
2. Select **Master (MAC) Mode - 2 IXP, 1BSF, 2 IX Bus** from the **Template Name** box.

*Note:* The Workbench detects if the microcode tries to address a non-existent device on the IX Bus. If this happens, an error message appears and the simulation stops.

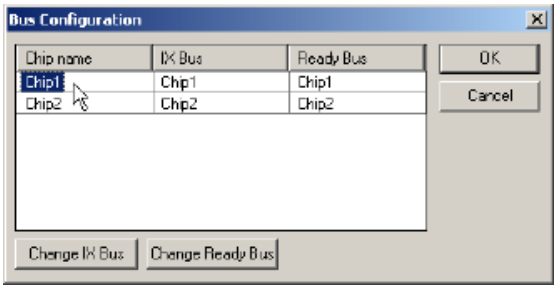
### 2.5.5.1 Customizing Buses for Multichip Configuration

By default, if you specify multiple chips, each chip is connected to its own IX Bus and Ready Bus.

To establish different connections:

1. On the **IX Bus** tab, click **No IXB3208 to configure**, and then click **Advanced IX Bus Configuration**.

   The **Bus Configuration** dialog box appears. The name for an IX Bus or Ready Bus must correspond to a chip name.




To change the IX Bus that a chip is connected to:

1. Select the chip.
2. Click **Change IX Bus**.

   The **Select IX Bus** dialog box appears.

3. Select the new IX Bus to connect the chip to and click **OK**.

*Note:* The IX Bus choices may exclude one or more chip names to prohibit invalid bus configurations.

To change the Ready Bus that a chip is connected to:

1. Select the chip and click **Change Ready Bus**.

   The **Select Ready Bus** dialog box appears.

2. Select the new Ready Bus to connect the chip to and click **OK**.

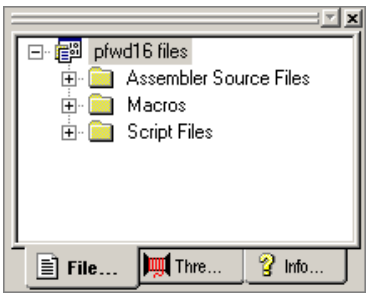
## 2.6 About the Project Workspace

The project workspace is a dockable window where you access and modify project configuration information (see Figure 2-1) files. It consists of three tabbed windows:

**FileView**

**ThreadView**

**InfoView**




To select a window, click its tab.

When you start the Workbench, only **InfoView** is visible.

When a project is open, **FileView** and **ThreadView** become visible, but access to **ThreadView** is unavailable.

When you start debugging, access to **ThreadView** is enabled.

When you stop debugging, access to **ThreadView** is disabled.

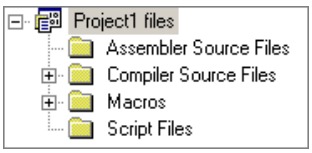
To toggle the visibility of the **Project Workspace**:

- On the **View** menu, select or clear **Project Workspace,** or

  Click the  button on the **View** toolbar.

### 2.6.1 About FileView

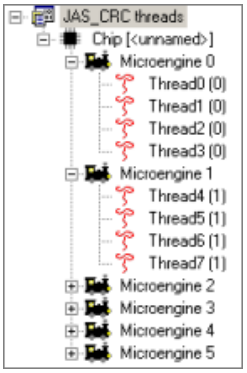
**FileView** contains a tree listing your project files. The top-level item in the tree is labeled <project-name> files. There are four second-level folders:




| | |
|---|---|
| **Assembler Source Files** | Expands to an alphabetical list of all project Assembler source files. |
| **Compiler Source Files** | Expands to an alphabetical list of all project Compiler source files. |
| **Macros** | Expands to list the macros that are defined in the project's source files. This folder expands to: **Macros saved by name**, and **Macros saved by file**. |
| **Script Files** | Expands to an alphabetical list of all debugging script files. |

## 2.6.2 About ThreadView

**ThreadView** lists a tree containing all Microengine threads. **ThreadView** provides access to all 24 threads for each IXP12nn Network Processor.




The top-level item in the tree is labeled <project-name> threads. There is a second-level item for each chip in the project. Each chip item expands to list the six Microengines in the chip. Each Microengine item expands to list the four threads in a Microengine. By default, a chip's threads are named Thread 0 through Thread 23. The Microengine number appears in parenthesis after the thread name.

### 2.6.2.1 Expanding and Collapsing Thread Trees

You can expand the entire tree for a chip as follows:

1. Right-click the chip name.
2. Click **Expand All** from the shortcut menu.

To collapse a chip's tree, double-click the chip name.

### 2.6.2.2 Renaming a Thread

You can rename a thread (to indicate its function or for any other reason. To do this:

1. Right-click the thread name in **ThreadView**.
2. Click **Rename Thread** from the shortcut menu.

   The **Rename Thread** dialog box appears.
3. Type the new name for the thread.
4. Click **OK**.

## 2.6.3 About InfoView

InfoView provides access to documentation on Workbench components:

Development Tools User's Guide

Programmer's Reference Manual

To view a document, double-click its name or icon. This invokes Adobe Acrobat Reader*, which then displays the document. A copy of Acrobat Reader is provided on the distribution CD-ROM.

## 2.7 Working with Files

The Workbench allows you to:

| | |
|---|---|
| **Create** files. | See Section 2.7.1. |
| **Open** files. | See Section 2.7.2. |
| **Close** files. | See Section 2.7.3. |
| **Save** files. | See Section 2.7.4. |
| **Save copies** of files. | See Section 2.7.5. |
| **Save all** files at once. | See Section 2.7.6. |
| **Print** files. | See Section 2.7.8. |
| **Insert** files into a project. | See Section 2.7.9. |
| **Remove** files from a project. | See Section 2.7.9. |
| **Edit** a file. | See Section 2.7.10. |
| **Bookmarks, error/tags.** | See Section 2.7.11. |

See also:

| | |
|---|---|
| **Working with File Windows.** | See Section 2.7.7. |
| **About Find in Files.** | See Section 2.7.12. |
| **About Fonts and Syntax colors.** | See Section 2.7.13. |
| **About Macros.** | See Section 2.7.14. |

### 2.7.1 Creating New Files

To create a new file:

1. On the **File** menu, click **New**, or
   Click the  button on the **File** toolbar.
   The **New** dialog box appears.
2. Select which type of file you want to create from the list.
3. Click **OK**.

This creates a new document window. The name of the window in the title bar reflects the type of file you have created.

### 2.7.2 Opening Files

To open an file for viewing or editing, do one of the following:

- On the **File** menu, click **Open**, and select a file from the **Open** dialog box, or
  Click the  button on the **File** toolbar, or
  If the file is in your project, double-click the file in **FileView**.

In the open dialog box you can filter your choices using the **Files of type:** list to select a file extension. This limits your choices to only files with that extension. If you select All files (*.*), your choices are unlimited. You can select any unformatted text file to view or edit.

You can open any of the last four files that you have opened. To do this:

1. On the **File** menu, click **Recent Files**.
2. Select from the list of files that appears to the right.

## 2.7.3 Closing Files

To close an open file:

- On the **File** menu, click **Close**, or

  On the **Window** menu, click **Close** to close the active file and its document window, or

  On the **Windows** menu, click **Close All** to close all open files and their document windows.

***Note:*** All files that have been edited but not saved are automatically saved when you perform any operation that uses file data, such as assembling, building, updating dependencies, and finding in files.

## 2.7.4 Saving Files

To save a file:

1. On the **File** menu, click **Save**, or

   Click the  button on the **File** toolbar.

   If you have just created the new file, the **Save As** dialog box appears. If you are saving an existing file, the **Save** dialog box appears.
2. Type the name of the new file.
3. Click **OK**.

   This saves your work to a file when you are finished editing. It also displays the new file name in the title bar of the window. By convention, microcode source files have the file type .uc, C Compiler source files have the file type .c, and script files have the file type .ind.

   If you are saving an existing file, you do not need to type a new name.

To save a file under a new name:

1. On the **File** menu, click **Save As**.

   The **Save As** dialog appears. The current name of the file appears in the **File Name** box.
2. Type a new name in the **File Name** box and click **Save**.

   Note that the old file remains in the folder but will not have edits that you have made. The new name appears in the title bar.

## 2.7.5 Saving Copies of Files

You can save a copy of a file that you are viewing or editing.

To do this:

1. On the **File** menu, click **Save As**.

   The **Save As** dialog box appears.

2. Browse to the folder where you want to save the file.
3. Type the new name of the file in the **File name** box.
4. Click **Save**.

The **Save as type** list is used only if you don't include the extension in the **File name** box. If you select All files (*.*), you must include the extension in the name.

## 2.7.6 Saving All Files at Once

You can save all modified files in your project at once.

To do this:

- On the **File** menu, click **Save All**, or

  Click the  button on the **File** toolbar.

## 2.7.7 Working With File Windows

When you select a file (text, Assembler source, Compiler source, source header, or script) for viewing or editing, it appears in a file window in the upper-left part of the Workbench. The **Windows** menu deals mostly with the text file windows in the Workbench.

| | |
|---|---|
| **New Window** | Creates a new window containing a copy of the file in the active window. The Title Bar displays *filename.ext*:2. |
| **Close** | Closes the active window. |
| **Close All** | Closes all the open windows. |
| **Cascade** | Cascades all windows that are not minimized in the viewing area. |
| **Tile** | Tiles all windows that are not minimized in the viewing area. |
| **Arrange Icons** | Tiles the window icons (if minimized) at the bottom of the viewing area. |


### Open Windows Selection

At the bottom of the **Windows** menu is a list of the first nine windows that you opened. Click any one of these windows to make it the active window. If you opened more than nine windows, click **More Windows**. From the **Select Window** dialog box, click the window that you want to make active and then click **OK**.

### Other Window Controls

| | |
|---|---|
| **Minimize** | Click the button on the window that you want to minimize. |
| **Maximize** | Click the button on the window that you want to maximize. You can also double-click the title bar to do this. |
| **Close** | Click the button on the window that you want to close. |
| **Restore** | Click the button on the minimized window that you want to restore to its previous view. |

## 2.7.8 Printing Files

### 2.7.8.1 Setting Up the Printer

1. On the **File** menu, click **Printer Setup**

   The **Print Setup** dialog box appears.

2. Select the printer properties for your printer. They will vary depending on the printer you select in the **Name** box.

3. Click **OK** when done.


Setting the printer properties does not print the file. To do this see Section 2.7.8.3.

### 2.7.8.2 Previewing the Printed File

1. On the **File** menu, click **Print Preview**.

   The **Print Preview** window appears with the name of the file you are viewing in the title bar.

2. Click **Two Page** to view two pages at a time.

3. Click **Zoom In** and **Zoom Out** to magnify the preview.

4. Click **Next Page** or **Previous Page** to browse through your file.

5. Click **Print** if you would like to print the file. The **Print** dialog box appears. This is explained in Section 2.7.8.3.

6. If you do not want to print the file, click **Close**.

### 2.7.8.3 Printing the File

You can print text files to a hardcopy printer or to a file.

To do this:

1. Make sure that the file you want to print is in the active window.

2. On the **File** menu, click **Print**, or

   Click the button on the **File** toolbar. (This button is not on the default **File** menu. To put this button there, see Section 2.3.3.4.)

   The **Print** dialog box appears.

3. Select the printer (or printer driver) from the **Name** list.
4. Click **Properties** to customize your particular printer. Each printer has its own printer settings.
5. If you want to print to a file (*.prn), select **Print to file** and select a folder and file name after you click **Print**.
6. Select the pages you want to print in the **Print range** area.
7. Select the number of copies in the **Copies** area.
8. Click **Print**.

## 2.7.9 Inserting Into and Removing Files from a Project

### 2.7.9.1 Inserting Files Into a Project

You can insert Assembler source files, Compiler source files, and script files into a project.

To do this:

1. On the **Project** menu, click **Insert Assembler Source File**, or

   **Insert Assembler Source File**, or

   **Insert Script Files**, whichever is appropriate.

   The corresponding dialog box appears.
2. Browse to the desired folder and select one or more files to be inserted.
3. Click **Insert**.

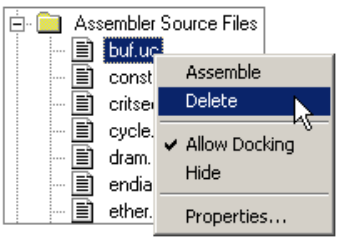
The newly inserted files are added to the list of files displayed in **FileView** in the corresponding folder.

### 2.7.9.2 Removing Files From a Project

To remove a file from your project:

1. In the **Project Workspace**, click the **File View** tab.
2. Right-click the file that you want to delete.
3. Click **Delete** on the shortcut menu, or

   Select the file and then press the DELETE key.




***Note:*** The file is removed from the project but it is not deleted from the disk.

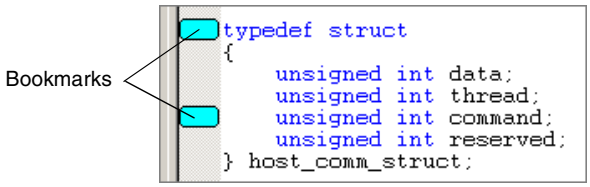
## 2.7.10 Editing Files

The Workbench editor is similar to standard text editors. See Table B-5 on page B-2 for a list of supported actions and their shortcuts.

If a file has been modified, an asterisk appears after its name in the Workbench title bar.

## 2.7.11 Bookmarks and Errors/Tags

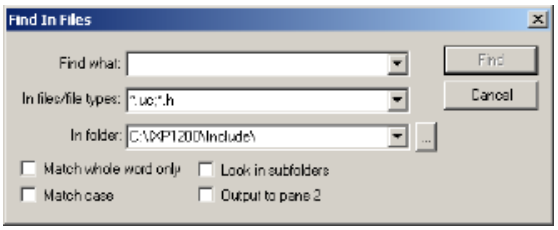
You can mark your place in a file using bookmarks. The table below lists the tools to manipulate bookmarks in your files.




You can find errors in your files using the Error/Tag tools listed in Table B-6 on page B-3 and Table B-9 on page B-4.

## 2.7.12 About Find In Files

The Workbench supports the ability to search multiple files for the occurrence of a specified text string. To perform this search:




1. On the **Edit** menu, click **Find In Files**, or

   Click the button on the **Edit** toolbar.

   The **Find In Files** dialog box appears.

2. Type the text string you want to search for, or select from the list of previously searched-for strings from the **Find what** list.

3. Type the file types to be searched, or select from a predefined list of file types in the **In files/file types** list.

   This box acts as a filter on the names of files to be searched. For example, you can specify “foo*.txt” to search only files with names that begin with “foo” and have an file extension of “txt”.

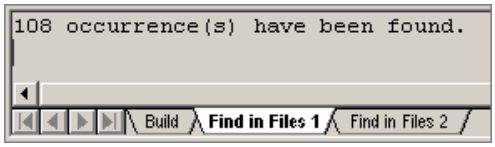
4. Type the name of the folder to be searched in the **In folder** box, or select from the list of previously searched folders in the list. You can also browse for the folder by clicking the ... button to the right of the **In folder** box.

5. You can also select from the options:

   | | |
   |---|---|
   | **Match whole word only** | Search for whole word matches only. The characters (){}[]"'<>,.?/\;$#@!~+=-|:*&^%, plus space, tab, carriage return and line feed are considered delimiters of whole words. |
   | **Match case** | Search only for strings that match the case of the characters in your string. |
   | **Look in subfolders** | Search all subfolders beneath the specified folder. |
   | **Output to pane 2** | Display the search results in the second output pane, labeled **Find In Files 2**. |

6. When you have selected all the options, click **Find**.

The results of the search are displayed in the **Find In Files 1** (or 2) tab of the **Output** window. For each occurrence of the search string that is found, the file name, line number, and line of text are displayed.

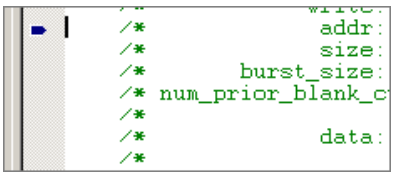



Do any of the following to display an occurrence of the search string:

- Double-click the occurrence.
- Click the occurrence and then press ENTER.
- Press F4 (the default key binding for the GoToNextTag command) to go to the next occurrence. If no occurrence is currently selected, then the first occurrence becomes selected. If the last occurrence is currently selected, then no occurrence is selected, or
- Press SHIFT+F4 to go to the previous occurrence. If no occurrence is currently selected, then the last occurrence becomes selected. If the first occurrence is currently selected, then no occurrence is selected.

In all cases, the window containing the file is automatically put on top of the document windows. If the file isn't already open, it is automatically opened.

The line containing the occurrence is marked with a blue arrow.




## 2.7.13 About Fonts and Syntax Coloring

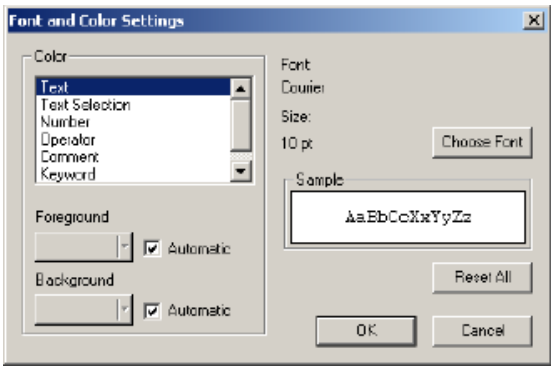
Source files, that is, those with file extensions of .uc, .c or .h, appear with syntax coloring of keywords and comments. Keywords are words that are reserved by the Assembler and Compiler are used in specific context. For example, 'alu_shf' is reserved because it is an Assembler instruction.

Comments comprise ';' followed by text on a line in Assembler language. By default, keywords are colored blue and comments are colored green.

To change color defaults:

1. Open a source file.
2. On the **Edit** menu, click **Options**.

   The **Font and Color Settings** dialog box appears.
3. In the **Color** list box, select the item for which you want to specify a color.




At the **Foreground** and **Background** controls, the colors already selected for the item you selected in Step 3 are displayed.

- Select **Automatic** to use the Window's default colors.
- Clear **Automatic** to enable the color selection controls. Then select a color for the item you selected.

Continue this procedure for any other items that you want to change.

- To change fonts, click **Choose Font** to select a different font for display.
- To go back to original settings, click **Reset All**.

  Your customized settings are saved in the UcSyntaxColoring.ini file located in the folder with the Workbench executable.

### 2.7.14 About Macros

The **FileView** tab in the **Project Workspace** has a Macro folder that contains the macros that are defined in the project's source files.

The macros are:

- Listed alphabetically, in the **By Name** folder, and
- Grouped according to the file that they are defined in, in the **By File** folder.

The Workbench:

- Creates these folders when you open a project.
- Updates them when:
  - An edited source file is saved,
  - A source file is inserted into or deleted from the project, or
  - You update dependencies.

To go to the location in the source file where a macro is defined, double-click the macro name.

If an opened source file contains a macro reference and you want to go to the file and location where that macro is defined:

1. Right-click the macro reference.
2. Click **Go To Macro Definition** on the shortcut menu.

## 2.8 About the Assembler

The Workbench contains an Assembler for your *.uc source files.

For information on:

| | |
|---|---|
| **Creating new Assembler source files**. | See Section 2.7.1. |
| **Saving Assembler source files**. | See Section 2.7.4. |
| **Opening Assembler source files**. | See Section 2.7.2. |
| **Editing Assembler source files**. | See Section 2.7.10. |
| **Closing an Assembler source file**. | See Section 2.7.3. |
| **Searching for text in an Assembler source file**. | See Sections 2.7.10 and 2.7.12. |
| **Fonts and syntax colors in an Assembler source file**. | See Section 2.7.13. |

The following topics on the Assembler will help you understand:

| | |
|---|---|
| **Root Files and Dependencies**. | See Section 2.8.1. |
| **Assembler Build Settings**. | See Section 2.8.2. |
| **Specifying Additional Include Directories**. | See Section 2.8.2.1. |
| **Specifying Processor Revision Range**. | See Section 2.8.2.2. |
| **Specifying Assembler Options**. | See Section 2.8.2.3. |
| **Invoking the Assembler**. | See Section 2.8.3 |
| **Assembly Errors**. | See Section 2.8.4 |
| **Invoking the Assembler**. | See Section 2.8.3 |

## 2.8.1 About Root Files and Dependencies

The executable image for a Microengine is generated by a single invocation of the Assembler that produces an output '.list' file. You can place all the code for a Microengine into a single source file, or you can modularize it into multiple source files. However, the Assembler allows you to specify only a single filename. Therefore, to use multiple source files, you must designate a primary, or root, file as the one that gets specified to the Assembler. You include the other files from within the root file or from within already included files, by nesting or chaining them. The included files are considered to be descendants of the root file. In the **FileView** tab of the **Project Workspace**, root files are distinguished by having a  to the left of it.

You can designate the same output file to be loaded into more than one Microengine. You can also include the same source file under more than one root file, making the file a descendant of multiple root files.

In order for the Workbench to build list and image files, you must assign a .list file to each Microengine. You set root files as part of setting Assembler options.

On the **Project** menu, click **Update Dependencies** to have the Workbench update the dependencies for all source files in the project.

If a file is included by a source file but is not itself a source file in the project, the Workbench automatically inserts that source file into the project. The Workbench automatically performs a dependency update when a project is opened. When you insert a microcode file into a project, the Workbench checks that file for dependencies.

## 2.8.2 Selecting Assembler Build Settings

To make or change Assembler settings:

1. On the **Build** menu, click **Settings**.

   The **Build Settings** dialog box appears.

2. Click the **General** tab to specify additional include directories (see Section 2.8.2.1) and the processor revision range (see Section 2.8.2.2).

3. Click the **Assembler** tab to specify parameters for creating .list files and other Assembler options (see Section 2.8.2.3).

*Note:* Compiler settings on the **General** tab are covered in Section 2.9.2.

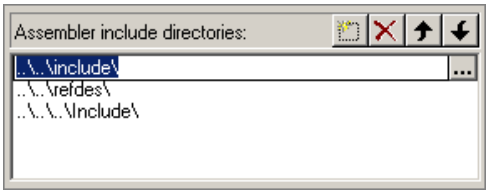
### 2.8.2.1 Specifying Additional Include Directories

The Assembler needs to know which folders contain the files referenced in `#include` statements in Assembler source files.

To do this:

1. On the **Build** menu**,** click **Settings**.
2. Click the **General** tab.

To specify additional Assembler include directories, the following controls are provided:




- A button to specify a new path.

  Type the path name in the space provided or use the browse button ... to search for it. You must double-click the include path listed in order to display the browse button.
- A button to delete an included path from the list.
- A button to move an included path up the list.
- A button to move an included path down the list.

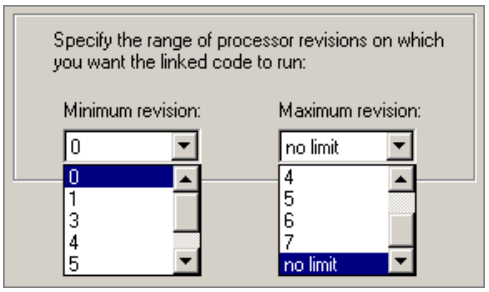
#### Absolute Versus Relative Paths

Regardless of whether the path information is entered in an absolute or relative format, it is automatically converted to a relative format. This allows the project to be moved to other locations on a system or to other systems without rendering the path information invalid in most instances as long as files are maintained in the same relative locations. This path information is passed to the Assembler so it may locate the files referenced in #include statements in the source code. It is also used by the dependency checker for locating assembly source files in the project.

### 2.8.2.2 Specifying Processor Revision Range

The IXP12nn network processors are available in different versions with different features. You can specify a range of revisions for which you want your microcode assembled. Section 3.1.5 covers this topic in more detail. Also, see Section 2.5.1.1 for chip revision information.

Do the following:




1. On the **Build** menu, click **Settings**.

   The **Build Settings** dialog box appears.
2. Click the **General** tab.
3. Select the range of processor revisions on which you want the linked code to run.

   Select from the **Minimum revision** and the **Maximum revision** lists. If you select **no limit** as the maximum revision number then you are specifying that your microcode is written to run on all revisions of the processor.

### 2.8.2.3 Specifying Assembler Options

To specify Assembler options, do the following:

1. On the **Build** menu**,** click **Settings**.

   The **Build Settings** dialog box appears.

2. Click the **Assembler** tab.

#### The .list File

The **Output to target .list file** box allows you to select a .list file from the set of .list files that are currently defined in the project. All other controls on the page are updated according to which .list file is selected in the box.

#### Inserting the .list File


1. On the **Assembler** tab, click **New**.

   The **Insert New List File into Project** dialog box appears.

2. Select a path for the .list file.
3. Type a filename.

   You cannot insert a .list file that has already been inserted into the project.

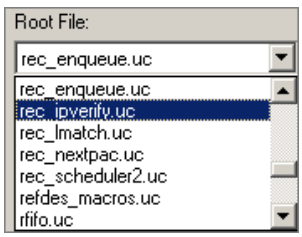
4. Click **Insert List File**.

   This closes the dialog box and adds the new filename to the list. The file's path appears in the read-only **Path of target .list file** list. The rest of the boxes assume default values.

#### Deleting the .list File

To delete a .list file from a project, click **Delete**. This removes the .list file currently selected in the list box from the project. All references to the file on the **Linker Build Settings** page are removed. The actual .list file, if it exists on disk, is not altered or deleted.

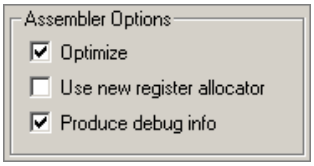
#### Root Files




The **Root File** list provides a read-only list of all of the .uc and .h files in the project. Select a file to designate it as the root file for the .list file.

If no root file is selected, "- no root file -" (default) is displayed. If a root file is not selected and you attempt to select another page or close the dialog with the **OK** button a warning message appears.

#### Optimizing




Select **Optimize** to turn on Assembler optimizations, or clear it to turn off optimizations.

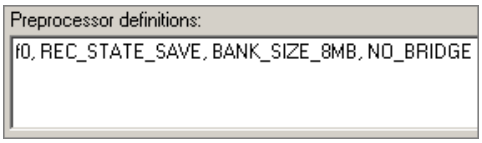
#### Use Of New Register Allocator

Select **Use New Register Allocator** to enable the Assembler to efficiently allocate registers, or clear it to rely on the `.local` directives for allocating registers.

### Produce Debug Information

Select **Produce debug info** to add debug information to the output file. If you do not select this option, you will not be able to open a thread window in debug mode.

### Preprocessor Definitions




Preprocessor definitions are symbols used in `#ifdef` and `#ifndef` statements to conditionally assemble sections of source files. Multiple definitions are separated by spaces. Optionally a replacement value may be assigned to a definition by append an "=" and a value; no spaces can occur between the symbol name and the "=" or between the "=" and the value. Default is blank.

### Edit/Override

The **Edit/Override** box allows you to change the Assembler command line parameters in the **Parameters used to invoke assembler** box. The default is disabled. Use this to fine tune your build settings.

### Save Build Settings

The **Build Settings** dialog box works with a copy of the build settings in the project. When you click **OK**, the Workbench validates your data and does not allow the dialog box to close if there are any errors. Validation is independent of which page is active at the time.

You have the option to fix the errors or click **Cancel** if you choose not to save any changes you have made. When the data in **Build Settings** passes validation, the data in the project is updated.

## 2.8.3 Invoking the Assembler

To assemble a microcode source file:

1. On the **File** menu, click **Open** to open the file or double-click on the file in **FileView**.

   If the file is already open, activate its document window by clicking on the file window.

2. On the **Build** menu, click **Assemble,** or

   Press CTRL+F7, or

   Click the  button.

### Root Files

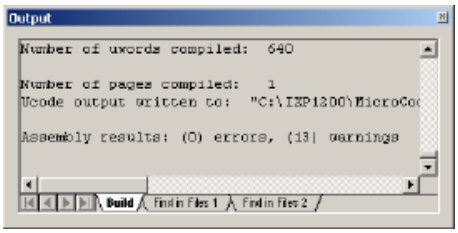
If the file is a root file, the Workbench assembles it using the Assembler settings associated with the Microengine for which the file is a root. Otherwise, the Workbench determines which root files the file is a descendant of. Therefore:

- If only one root file is found, the Workbench assembles that root file using the associated Assembler settings.
- If multiple root files are found, the Workbench displays a dialog box listing the root files. Select one or more of these root files to be assembled.

- If the active file is neither a descendant nor a root file, the Workbench displays a dialog box informing you of that fact and requests confirmation to proceed with the assembly using the default Assembler settings.

### Results

The results of an assembly appear in the **Build** tab of the **Output** window, which appears automatically.




You can control the amount of detail provide in the results. On the **Build** menu, click **Verbose Output** to toggle between getting detailed results and summary results.

Assembly is also done as part of a build operation.

*Note:* You can toggle the visibility of the **Output** window by clicking the  button on the **View** toolbar.

## 2.8.4 About Assembly Errors

Assembly errors appear in the **Build** tab of the **Output** window. Do any of the following to display the line of source code that caused an error:

- Double-click the error description, or

  Press F4, or

  Click the  button.

If no error is selected, the first error becomes selected. If the last error is selected, then no error is selected.

To go to the source line for the previous error:

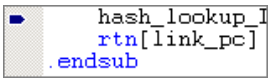
- Press SHIFT+F4, or

  Click the  button.

If no error is selected, then the last error becomes selected. If the first error is selected, then no error is selected.

In all cases, the window containing the source file is put on top of the document windows. If the source file isn't open, Workbench opens it.

A blue arrow in the left margin marks lines containing errors. Only one error at a time is marked.




*Note:* The default Debug toolbar does not contain these buttons. To add them, go to Section 2.3.3.4.

## 2.9 About the Microengine C Compiler

The Workbench contains a C Compiler to compile C source code into microcode for the Microengines. The Microengine C Compiler is a general purpose compiler but the C language used for the Microengines is limited. Refer to the *IXP1200 Microengine C Language Support Reference Manual* for information on the functions and intrinsics designed for use with the IXP12nn family.

For information on:

| | |
|---|---|
| **Creating** new C source files. | See Section 2.7.1. |
| **Saving** C source files. | See Section 2.7.4. |
| **Opening** C source files. | See Section 2.7.2. |
| **Editing** C source files. | See Section 2.7.10. |
| **Closing** a C source file. | See Section 2.7.3. |
| **Searching for text** in a C source file. | See Sections 2.7.10 and 2.7.12. |
| **Fonts and syntax colors** in a C source file. | See Section 2.7.13. |

This section details:

| | |
|---|---|
| **Adding C source files to your project**. | See Section 2.9.1. |
| **Selecting the target .list file**. | See Section 2.9.2.2. |
| **Selecting C source files to compile**. | See Section 2.9.2.3. |
| **Removing C source files from project**. | See Section 2.9.2.5. |
| **Selecting Compile options**. | See Section 2.9.2.6. |
| **Invoking the C Compiler**. | See Section 2.9.3. |
| **Compilation errors**. | See Section 2.9.4. |

### 2.9.1 Adding C Source Files to Your Project

After creating and saving C source files, you need to add them to your project. To do this:

1. On the **Project** menu**,** click **Insert Compiler Source Files**.

   The **Insert Compiler Source Files into Project** dialog box appears.
2. From the **Look in** list, browse to the folder containing your C source file(s).
3. Select the file(s) that you want to insert into your project.
4. Click **Insert**.

In the **Project Workspace** window, to the left of the **Compiler Source Files** folder, a '+' appears (if the folder was previously empty) indicating the folder now contains files. Click the '+' to expand the folder and display the files. You should see the files that you have just added to your project.

### 2.9.2 Selecting Compiler Build Settings

Before building you project, you must select your C Compiler options.

*Note:* Assembler settings on the **General** tab are covered in Section 2.8.2.

### 2.9.2.1 Selecting the Compiler Include Paths

The C Compiler needs to know which areas of the file system to search for locating files referenced in #include statements in C source code files. This control displays a list of paths with a GUI for typing in or editing of directory paths, or browsing to directories to be added to the list. The GUI also provides the means for deleting or changing the search order of the paths. Regardless of whether the path information is entered in an absolute or relative format, it is automatically converted to a relative format. This allows the project to be moved to other locations on a system or to other systems without rendering the path information invalid in most instances, so long as the relative location of the paths is maintained. This path information is passed to the Assembler so it may locate the files referenced in #include statements in the source code. It is also used by the dependency checker for locating C source files in the project.

To specify additional Compiler include directories, the following controls are provided:

- A  button to specify a new path.

  Type the path name in the space provided or use the  button to search for it. You must double-click the include path listed in order to display the browse button.
- A  button to delete an included path from the list.
- A  button to move an included path up the list.
- A  button to move an included path down the list.

### 2.9.2.2 Selecting the target .list file

When you compile your C source file, the results become a .list file. You must select the name of the .list file.

To do this:

1. On the **Build Settings** dialog box, click the **C Compiler** tab.
2. Select the name of the .list file from the **Output to target list file** list, or if the list is empty,
   a. Click **New**.
      The **Insert New List File into Project** dialog box appears.
   b. In the **Look in** list, browse to the folder where you want to store the .list file.
   c. Type the file name in the **File name** box.
   d. Click **Insert List File**.

The **Path of target .list file** box is a read-only text field displaying the absolute path of the target .list file. If this path is not correct, click **New** again and select a new path.

### 2.9.2.3 Deleting a Target .list File

To delete a target .list file from the project:

1. Select the file from the list in the **Output to target .list file** box.
2. Click **Delete**.

*Note:* This removes the file from the project but does not delete it from the disk.

### 2.9.2.4 Selecting C Source Files to Compile

The C Compiler in the Workbench can compile one or more C source files into one .list file. You must select the source files that you want to compile. To do this:

1. In the **Build Settings** dialog box, click the **C Compiler** tab.
2. Click **Choose source files**.

   The **Compiler Sources** dialog box appears. The files displayed here are all the *.c files in your project, that is all the files in the **Compiler Source Files** folder in the **Project Workspace** window.
3. Click the file(s) that you want to compile.

   Clicking the file once selects the file and clicking a selected file deselects it.
4. Click the  button to move the selected files from the left window to the right window.

   You can select any file(s) in the right window and move them back to the left window by clicking the  button.
5. Click **OK** when done.

In the **Source files to compile** box is a list of C source files that you selected to compile.

### 2.9.2.5 Removing C Source Files to Compile

To remove any file:

1. Click the desired file in the **Source files to compile** list.
2. Click the  button.

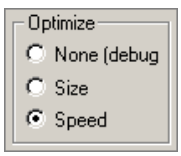
This removes the file from the compilation but not from the project.

### 2.9.2.6 Selecting Compile Options

In the **Compiler Options** box, select:

#### Optimize

| | |
|---|---|
| **None** (debug) | Turns off optimizations for better code troubleshooting. |
| **Size** | Compiled for smallest memory footprint. Speed may be sacrificed. |
| **Speed** (default) | Compiled for fastest instruction execution. Size may be sacrificed. |

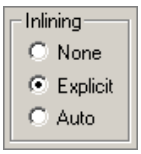


### Inlining

**None** — No inlining is done, including functions explicitly tagged in the source code with the `inline` specifier.

**Explicit** (default) — Only functions tagged with the `inline` specifier are inlined. Any function that could be inlined by the Compiler but not having this tag is not inlined.

**Auto** — All functions with the `inline` tag and all other functions thought by the Compiler to be inlinable are inlined.




### Endian Mode

**Little Endian** — Compile in little-endian mode.

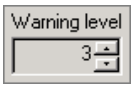
**Big Endian** (default) — Compile in big-endian mode.


### Warning Level

**0** — Print only errors.

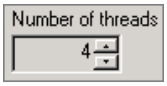
**1, 2, or 3** (default) — Print only errors and warnings.

**4** — Print errors, warnings, and remarks.




### Number of Threads

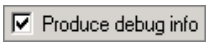
**1, 2, 3, 4** (default) — Select the number of threads that you want to be active in the Microengine. All others are killed.




### Produce Debug Information

**Select** (default) — Produces debug information in the .list file. This information is needed for many of the debugging features of the Workbench.

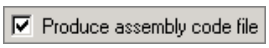
**Clear** — No debug information is compiled into the .list file.




### Produce Assembly Code File

**Select** — Produces an assembly code file (*.uc).

**Clear** (default) — Does not produce an assembly code file.




### Preprocessor Definitions

This is a text edit box where you type symbols used in `#ifdef` and `#ifndef` statements to conditionally compile sections of Assembler sources. Multiple definitions are separated by spaces. Optionally a replacement value may be assigned by appending an "=" and a value. There can be not spaces between the symbol name and the "=" or between the "=" and the value. The default is blank.

### 2.9.2.7 Edit/Override

The Edit/Override check box allows you to change the Compiler parameters in the **Parameters used to invoke compiler** box. The default is disabled. Selecting it allows you to change the information in the box. See Section 4.3, "Supported Option Switches."" for more information.

### 2.9.2.8 Saving Build Settings

The **Build Settings** dialog box works with a copy of the build settings in the project. When you click **OK**, the Workbench validates your data and does not allow the dialog box to close if there are any errors. Validation is independent of which page is active at the time. You have the option to fix the errors or click **Cancel** if you choose not to save any changes you have made. When the data in **Build Settings** passes validation, the data in the project is updated.

## 2.9.3 Invoking the Compiler

To compile a C source file:

1. On the **File** menu, click **Open**, or

   You can also double-click the file in FileView. If the file is already open, activate its document window by clicking on the file window.

2. On the **Build** menu, click **Compile**, or

   Press CNTRL+SHIFT+F7, or

   Click the  button on the **Build** toolbar.

#### Results

The results of an assembly appear in the **Build** tab of the **Output** window, which automatically appears.

You can control the amount of detail provide in the results. On the **Build** menu, select **Verbose Output** to display detailed results, or clear it to display summary results.

Compilation is also done as part of a build operation.

## 2.9.4 Compilation Errors

Compiler errors appear in the **Build** tab of the **Output** window.To locate the error in the source file:

- Double-click the error description in the **Output** window, or

  Click the error description, then press ENTER.

You can press F4 or click the  button to go to the next error. If no error is selected in the **Output** window, the first error becomes selected. If the last error is selected, then no error is selected, or

You can press SHIFT+F4 or click the  button to go to the previous error. If no error is selected in the **Output** window, the last error becomes selected. If the first error is selected, then no error is selected.

In all cases, the window containing the source file is put on top of the document windows and becomes the active document. If the source file isn't already open, it opens.

A blue arrow in the left margin marks lines containing errors. Only one error at a time is marked.

## 2.10 About the Linker

### The Linker

You build a project using the Linker. The Linker takes the Assembler or Compiler output (.list files) on a per-Microengine basis and generates an image file for up to six Microengines. To invoke the Linker and build a project:

- On the **Build** menu, click **Build**, or

  Click the  button on the **Build** toolbar, or

  Press F7.

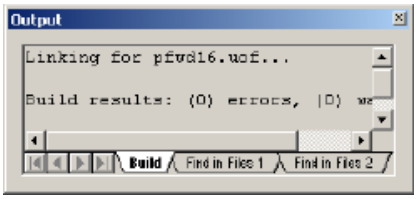
### Stopping Compilation

There is no way to stop a build in progress. You must wait until it finishes or encounters an error.

### Out-of-date Files

To perform a link, the Workbench requires that all .list files be up to date. If any microcode source file is newer than the list file generated from it or if Assembler settings have been changed since the last build, the Workbench automatically assembles a new .list file.

### Results




The results of the build appear in the **Build** tab of the **Output** window, which appears automatically. You can control the amount of detail provide in the results. On the **Build** menu, click **Verbose Output** to toggle between getting detailed results and summary results.

### Rebuild

To perform a full unconditional build of your configuration:

- On the **Build** menu, click **Rebuild,** or

  Press ALT+F7, or

  Click the  button on the **Build** toolbar.

## 2.10.1 Customizing a Build Configuration

To customize your build configuration:

1. On the **Build** menu, click **Settings**.

   The **Build Settings** dialog box appears.

2. Click the **Linker** tab to view the Linker settings.

3. Adjust the Linker settings (see Section 2.10.2).
4. Click **OK**.

The Linker page provides an interface for selecting options for the Linker and directing the packaging of one or more Microengine specific *.list files into a *.uof file. Each IXP12nn chip has 6 Microengines that can each be loaded with execution code according to the *.list file selected for that Microengine.

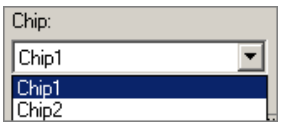
You can also specify the assembly options by clicking the **Assembler** tab in the **Build Settings** dialog box (see Section 2.8.2.3).

## 2.10.2 Changing Linker Settings

In the **Linker** page of the **Build Settings** dialog box, you can set the following build parameters:

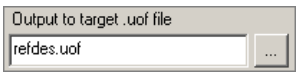
### Chip

The **Chip** box contains a list of all the IXP12nn chips in your project. Select the chip for which you want to change Linker settings. The other controls on the page are updated based on the selected chip.




### Output to Target .uof File

The **Output to target .uof file** box displays the .uof file that the Linker produces.




***Note:*** The Developer Workbench does not support multiple .uof files.

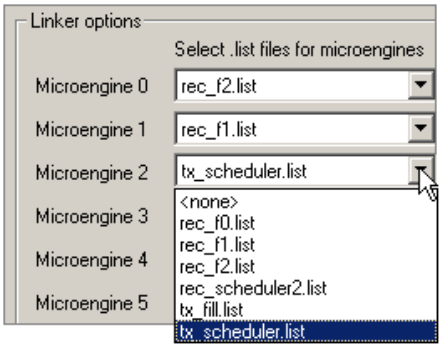
To change the output *.uof file to the project for the selected chip:

1. Click the ... button.

   The **Select Name and Location for the Linker Output File** dialog box appears.
2. In the **Look in** box, browse to the folder where you want to put the output file.
3. Type a new name in the **File name** box.

   You do not have to type the .uof extension—the Workbench adds it for you. Typing it does no harm.
4. Click **Select**.

### Microengine .list File Selection




The project has one or more .list file(s) generated using the Assembler or Compiler. On the **Linker** page you can control which .list file is linked into the .uof file and for which Microengine.

To do this:

1. View the list to the right of **Microengine 0**.
2. Select either:
   a. <none>, or
   b. Any .list file from list.
3. Do the same for **Microengines 1-5**.

This method allows you to select any combination of .list files or no .list file for any or all the Microengines to be linked to the .uof file.

If you specify <none>, no microcode is loaded into that Microengine. If you select <none> for all the Microengines, you get an error.

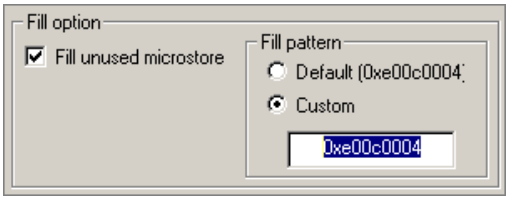
### Hex ".c" Files

Select **Generate hex '.c' file** to request the Linker to create a *.c file, with the same name as the corresponding *.uof file. This file contains a microcode listing in a form that you can include in an IXP processor core application. This is usually done when deploying microcode into a final product.

### Produce Debug Information

Select **Produce debug info** to add debug information to the output file. If you do not select this option, you will not be able to open a thread window in debug mode.

### Unused Microstore




Unused microstore can be initialized by using the controls in the **Fill option** area.

To leave the unused microstore unchanged, clear **Fill unused microstore**.

To fill the unused microstore:

1. Select **Fill unused microstore**.
2. Click **Default** to fill with 0xe00c0004, or

   Click **Custom** and type an eight character hex pattern to be used. Make sure the number begins with "0x."

### Reserved Memory Segment for Variables

The reserved memory segment for variables provides the Linker with information needed for allocating memory to be used for variable data storage.

### SCRATCH Base Address

The **Scratch base address** is a parameter sent to the Linker. The Linker uses scratch memory starting at the base address, allocating as much memory as needed up to the **Scratch max size** for variables.

### SCRATCH Max Size (Bytes)

The **Scratch max size** is a parameter sent to the Linker. The Linker reserves as much scratch memory as necessary for variables up to the max size.

### SRAM Base Address

The **SRAM base address** is a parameter sent to the Linker. The Linker uses scratch memory starting at the base address, allocating as much memory as needed up to the **SRAM max size** for variables.

### SRAM Max Size (Bytes)

The **SRAM max size** is a parameter sent to the Linker. The Linker reserves as much SRAM as necessary for variables up to the max size.

### SDRAM Base Address

The **SDRAM base address** is a parameter sent to the Linker. The Linker uses sdram memory starting at the base address, allocating as much memory as needed up to the **SDRAM max size** for variables.

### SDRAM Max Size (Bytes)

The **SDRAM max size** is a parameter sent to the Linker. The Linker reserves as much SDRAM as necessary for variables up to the max size.

### Header File Generation

Selecting **Generate a header file** causes the Linker to produce a C language *.h file with the same filename as the linked *.uof file. The defined symbols are set to values based on how the Linker allocated memory for the reserved memory variables. The base address symbols should have the same values as the ones defined in the GUI, but the size symbols have the actual sizes used by the Linker.

### Saving Settings

Linker settings are saved when you save the project.

### Build Settings

The **Build Settings** dialog box works with a copy of the build settings in the project. When you click **OK**, the Workbench validates your data and does not allow the dialog box to close if there are any errors. Validation is independent of which page on **Build Settings** is active at the time. You have the option to fix the errors or click **Cancel** if you choose not to save any changes you have made. When the data in **Build Settings** passes validation, the data in the project is updated.

# 2.11 Debugging

### Simulation Mode Versus Hardware Mode

Using the Workbench, you can debug microcode either in Simulation mode or in Hardware mode using the IXP12nn Evaluation Board or compatible hardware.

When in Simulation mode, the Transactor provides debugging support to the Workbench. In Hardware mode, the Microengine Debug Library (debug_1200.a) running as part of a StrongARM core application program communicates with the Workbench and relays debugging operations between the Workbench and the Microengines. The StrongARM core application may either be one that is supplied with the IXP12nn Evaluation Board or one that is independently developed.

The Workbench menus and toolbar selections provide the following capabilities:

- Set breakpoints and control execution of the microcode.
- View source code on a per-thread basis.
- Display the status and history of Microengines, threads, and queues.
- View and set breakpoints on data, registers, and pins.
- Configure and enable IX Bus device and network traffic simulation.

Some of the debugging operations are either disabled when debugging in Hardware mode or available in a limited fashion. The descriptions in the sections that follow include any limitations that apply in Hardware mode. Table 2-1 summarizes which debugging features are available in Hardware and Simulation modes.

**Table 2-1. Simulation and Hardware Mode Features (Sheet 1 of 2)**

| Feature | Simulation | Hardware |
|---|---|---|
| System Configuration | X | |
| Starting and Stopping Debug | X | X |
| Command Line Interface | X | X |
| Script Files | X | X |
| Command Scripts | X | X |
| Thread Windows | | |
| • Display Microword Address | X | X |
| • Instruction Markers | X | X |
| • View Instructions | X | X |
| Run Control | X | X[1] |
| Breakpoints | X | X[1] |
| Examine Registers | X | X |
| Watch Data | | |
| • Enter New Data Watch | X | X |
| • Watch CSRs and Pins | X | X[1] |
| • Watch GPRs and XFER | X | X |
| • Deposit Data | X | X[1] |

**Table 2-1. Simulation and Hardware Mode Features (Sheet 2 of 2)**

| Feature | Simulation | Hardware |
|---|---|---|
| • Break on Data Change | X | |
| Watch Memory | X | X |
| Performance Statistics | X | |
| Execution Coverage | X | |
| Thread and Queue History | X | |
| Queue Status | X | |
| Thread Status | X | X |
| IX Bus Device Status | X | |

**NOTES:**
1. With restrictions.

## 2.11.1 Starting and Stopping the Debugger

### Starting

To enter debug mode:

- On the **Debug** menu, click **Start Debugging**, or

  Press F12, or

  Click the  button.

Once the debugger begins, you can interact with it through the command line window and by using the **Debug** menu and toolbar selections that become activated.

### Stopping

To exit debug mode:

- On the **Debug** menu, click **Stop Debugging**, or

  Press CTRL+F12, or

  Click the  button.

Project debug settings such as breakpoints are automatically saved in a debug options file (.dwo) when you save a project.

## 2.11.2 Changing Simulation Options

### 2.11.2.1 Marking Instructions

You can select how instructions are marked in a thread window when a thread execution is stopped, such as at a breakpoint.

To modify the instruction marker:

1. On the **Simulation** menu, click **Options**.

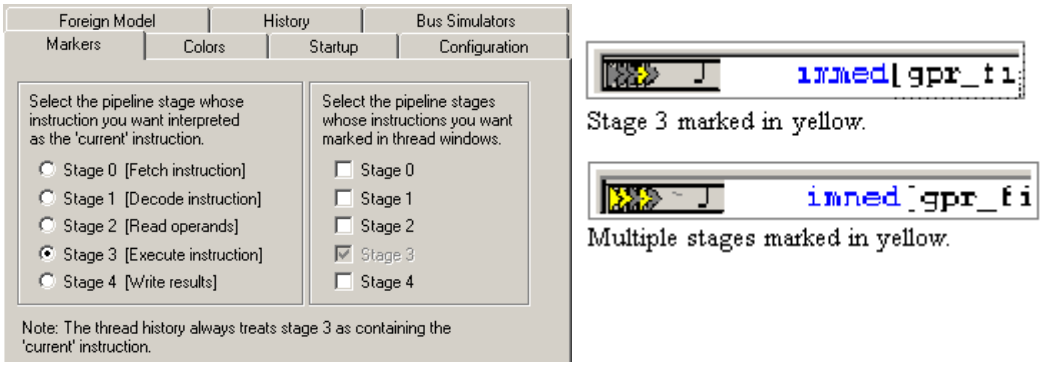
2. Click the **Markers** tab.

*Note:* For more information on thread windows, see Section 2.11.7.

By default, the stage 3 instruction is marked as the current instruction (see Figure 2-3). It is highlighted by horizontal black lines above and below it. The thread window is automatically scrolled so that the current instruction marker is visible when execution stops.

If the line containing the current instruction is displayed, then the instruction marker points to it. If the line is hidden because it is in a collapsed macro, then the instruction marker points to the line containing the collapsed macro.

**Figure 2-3. Marking Instructions**




You can optionally choose to have multiple instructions marked in addition to the current instruction when thread execution stops. The Workbench marks any combination of instructions that are in one of the 5 pipeline stages. To add the stages you want marked, click the appropriate check boxes in the Markers tab.

For more information on instruction markers, see:

Section Section 2.11.2.2, “Changing the Colors for Execution State.”

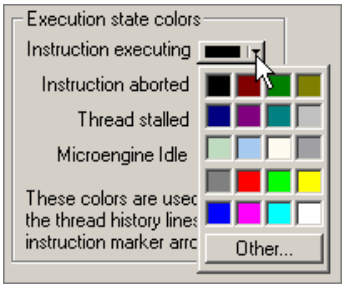
Section Section 2.11.7.9, “About Instruction Markers.”

Section Section 2.11.7.10, “Viewing Instruction Execution in the Thread Window.”

*Note:* You cannot clear the stage that you have selected for the current instruction.

### 2.11.2.2 Changing the Colors for Execution State

To customize the colors used to indicate the execution state:




1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.

2. Click the **Colors** tab.

3. Select the color for each execution state using the corresponding list.

   For more color options, click **Other**. Select the color you want and click **OK**.

4. Click **OK** when done.

*Note:* The execution state colors are used for both the Pipe Stage markers and for the thread history lines.

### 2.11.2.3 Initializing Simulation Startup Options

When you are debugging in Simulation mode, the Transactor and its hardware model must be initialized before you can run microcode.

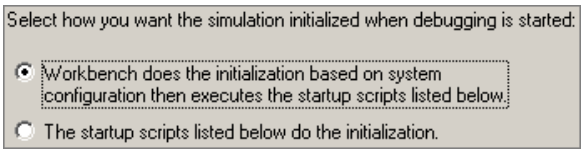
1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.

2. Click the **Startup** tab.

This property page specifies how the Workbench behaves when you start debugging and when you reset the simulation.

#### Initialization Customization




By default, all initialization is done by the Workbench. You can customize the initialization by selecting the Configuration tab.

On this page you can specify:

| | |
|---|---|
| The simulation step unit. | See Section 2.11.2.4. |
| The tagging of memory. | See Section 2.11.2.5. |
| The enabling of the PCI unit. | See Section 2.11.2.7. |
| The enabling of the StrongARM core model. | See Section 2.11.2.6. |

#### Startup Scripts

You can perform all initialization yourself. This requires you to do the initialization in a script file.

To have the Workbench execute one or more scripts at startup:

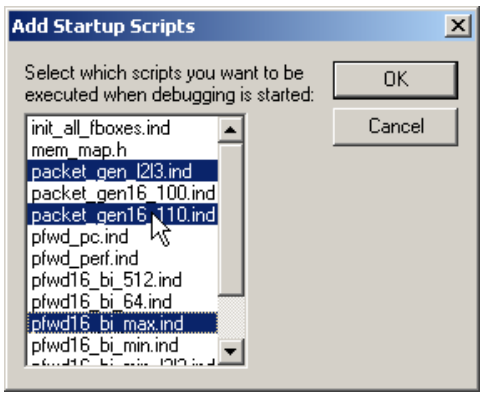
1. On the **Startup** tab, click **Add Script(s)**.

   The **Add Startup Scripts** dialog box appears.

2. Select the script(s) that you want executed at startup.

   Note that the scripts must be part of your project (in the Script File folder) to appear in this list. Otherwise the list is blank.

3. Click **OK** when done.




The Workbench executes the scripts in the order in which they appear in the **Scripts to be executed when debugging is started** box. You can change the order in which scripts are executed by selecting the script and using the **Up** and **Down** arrow buttons. You can delete a script from the list with the **Delete** button.

These scripts are executed even if the Workbench performs the initialization. If you specify that the Workbench perform model initialization, it will execute a series of commands in the Transactor, which do the following:

1. Define the chip using the chip command.
2. Initialize SRAM, SDRAM, and FLASH based on the sizes you specified when you created the project. This is done using the `mem_init sram`, `mem_init sdram`, and `mem_init flash` commands.
3. Initialize the model using the init command.
4. Load the microcode into all Microengines that are included in the Build Configuration.
5. Set the clock frequencies using the `set_clk_freq` command. The frequencies are those that you specified in the **Chip** tab in the **System Configuration** dialog box.
6. Configure the SRAM and SDRAM units using the `sram_config` and `sdram_config` commands. The configuration is based on values that you specified in the **Memory** tab in the **System Configuration** dialog box.
7. Configure the IX Bus Interface by depositing into several control and status registers. The configuration is based on values that you specified in the **IX Bus Interface** tab in the **System Configuration** dialog box.
8. Set the clock domain default using the `go_clk_domain` command.
9. Enable each Microengine included in the build configuration (see Sections 2.8.2 and 2.10.1). This is done using the `set dep.art.enable = 1` command, where n is the Microengine number.
10. Enable all four contexts in each enabled Microengine. This is done using the `dep/s fn.ctl.thread_trigger_mask_x` command, where n is the Microengine number and x is the context number.
11. Enable pipeline abort control in each enabled Microengine. This is done using the `dep/s fn.ctl.px_abort = 1` command, where n is the Microengine number and x is the pipeline stage.

If you enable the StrongARM model, do not perform steps 4, 9, 10, and 11.

#### Preferred Initialization Method

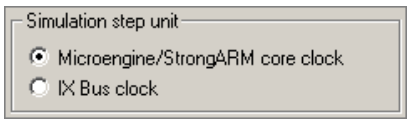
The suggested method of initialization is to let the Workbench perform the initialization as described above and to perform all other customized initialization using one or more startup scripts. Your startup scripts should avoid any conflicts with the initialization done by the Workbench.

### 2.11.2.4 Setting the Simulation Step Unit

The Workbench allows you to specify which clock domain is used when you single step or go for a specified number of cycles. To set the simulation step unit:

1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.
2. Click the **Configuration** tab.
3. In the **Simulation step unit** area, click **Microengine/StrongARM core clock** or **IX Bus clock**.



If you set the step unit before you start debugging, your selection takes effect only if you also elect to have the Workbench perform initialization (see Startup Options, Section 2.11.2.3"). After debugging has been started, changing the simulation step unit takes effect immediately.

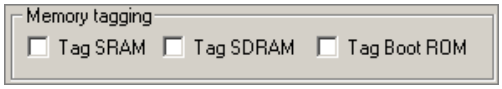
### 2.11.2.5 Tagging Memory

The Transactor supports the tagging of SDRAM, SRAM, and flash memory. This provides detailed information on access to each memory location. However, it consumes significantly more memory and slows down the simulation. If you have the Workbench do the initialization (see Startup Options, Section 2.11.2.3), you can specify memory tagging by selecting the appropriates check box(es) for the memory types.

To turn tagging on or off:

1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.




2. Click the **Configuration** tab.
3. Select or clear **Tag SRAM**, **Tag SDRAM,** or **Tag Flash Memory**.

### 2.11.2.6 Enabling StrongARM* Core Model

If you have the Workbench do the simulation initialization (see Startup Options, Section 2.11.2.3), you can conditionally enable the simulation of the StrongARM core model in order to debug your StrongARM boot code.

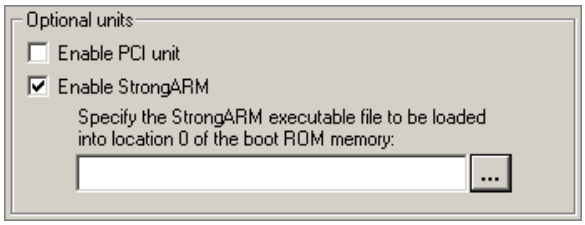
To enable StrongARM:

1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.

2. Click the **Configuration** tab.
3. Select or clear **Enable StrongARM**.

   If you enable the StrongARM model, you must specify the name of the file containing the image to be loaded into boot memory. This is the code that the StrongARM core executes.




*Note:* When you start debugging with the StrongARM core model enabled, eight simulation cycles are executed in order to perform initialization. Also, it is assumed that microcode loading is being done by the boot code, therefore, the Workbench does not load the microcode. However, the Workbench defines a Transactor function named `WB_Load_Microcode()` that you can invoke from a script at the appropriate time in order to do a quick loading of the microcode. This function executes Transactor `load_uc` commands to load the appropriate .list files as specified by the build settings.

### 2.11.2.7 Enabling the PCI Interface

To enable the simulation of the PCI Interface in the IXP12nn chip:

1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.
2. Click the **Configuration** tab.
3. Select **Enable PCI unit**.

To specify the configuration of the PCI Interface, see Section 2.5.4.

### 2.11.2.8 Enabling the PCI Transactor

When you are debugging in Simulation mode and you enable the PCI Interface, you can simulate one or two generic PCI devices. To do this:

1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.
2. Click the **Bus Simulators** tab.

   Here you can:

- Enable the simulation one or both of the PCI devices, and
- Specify the start address and size for each.

### 2.11.2.9 Enabling IX Bus Device Simulation

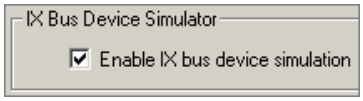
When you are debugging in Simulation mode and you enable the IX Bus device simulation, you can:

- Simulate devices on the IX Bus as well as
- Network traffic coming into those devices.

To do this:

1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.
2. Click the **Bus Simulators** tab.
3. Select **Enable IX bus device simulation**.




## 2.11.3 Exporting the Startup Script

To create a text file containing all the commands that the Workbench sends to the Transactor during simulation startup:

1. On the **Simulation** menu, click **Export Startup Script**.

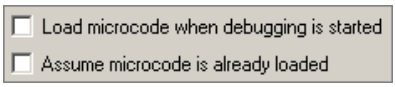
   The **Export Simulation Start Script** dialog box appears.
2. Browse to the folder to save the file.
3. Type the name of the script file in the **File name** box.
4. Click **Save**.

The default .ind file extension for script files is added to the name that you typed. See Section 2.7.9.1 to insert the script file into the project.

## 2.11.4 About Hardware Options

### 2.11.4.1 Specifying Hardware Startup Options

To specify whether or not the Workbench loads microcode when hardware debugging is started:




1. On the **Hardware** menu, click **Options**.

   The **Hardware Options** dialog box appears.
2. Click the **Startup** tab.
3. Select or clear **Load microcode when debugging is started**.
4. Click **OK**.

If you choose not to have the Workbench load microcode automatically at startup, you can:

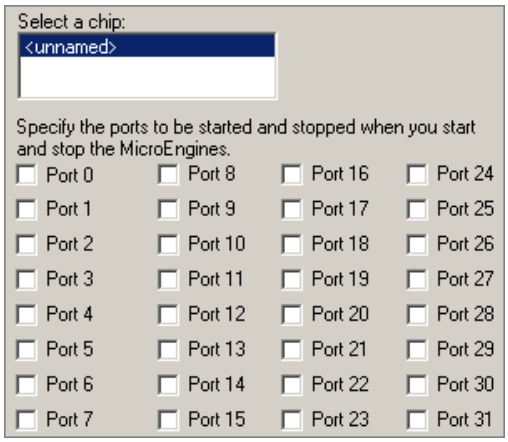
- Load microcode manually by selecting **Load Microcode** on the **Debug** menu.

  (You can also click the  button. This button is not on the default **Build** menu. To put this button there, see Section 2.3.3.4.)
- By selecting or clearing **Assume microcode is already loaded** in the **Hardware Options** dialog box, you can specify whether or not you want the Workbench to assume that microcode has been loaded into the Microengines by some other mechanism.

  Note that if you select this option, the Workbench connects to the hardware without resetting the Microengines.

### 2.11.4.2 Specifying MAC Port Control

To specify which MAC ports get started and stopped along with a chip's Microengines:



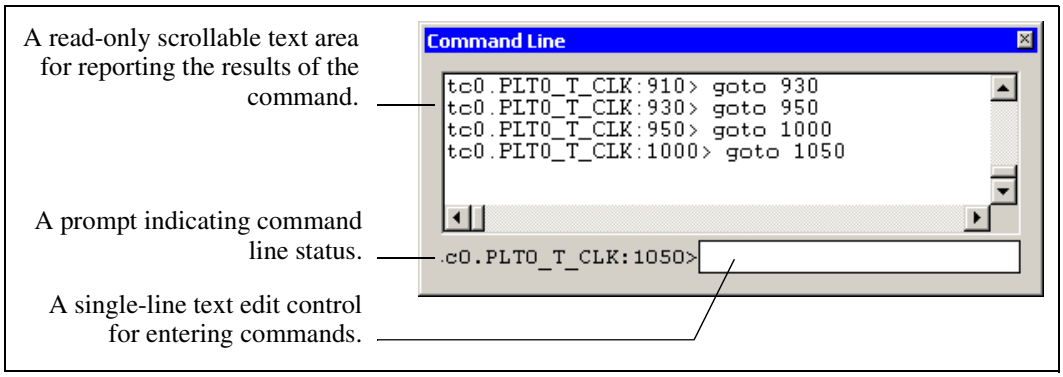

1. On the **Hardware** menu, click **Options**.

   The **Hardware Options** dialog box appears.
2. Click the **Ports** tab.
3. Select a chip from the **Select a chip** list if there is more than one available.
4. Select each **MAC port** that you want to start and stop along with the chip's Microengines.

   If there is more than one chip in your project, repeat steps 3 and 4 for the remaining chip(s).
5. Click **OK**.

## 2.11.5 About the Command Line Interface

The command line interface (CLI) comprises:




### Simulation Mode

If you are debugging in Simulation mode, the command line is an interface to the Transactor command line. Commands entered on the command line are passed to the Transactor. The command and the Transactor responses are logged into the command line output area. Also, when you perform a simulation operation using GUI controls, the Workbench sends the appropriate command to the Transactor, as if you had typed in on the command line.

### Hardware Mode

If you are debugging in Hardware mode, the command line provides an alternative way to access the hardware. The hardware command line supports C-interpreted functions, script files and conditional directives. Type `help` on the command line to get a complete list of supported commands.

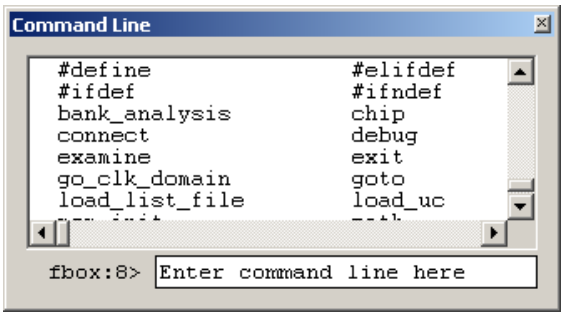
### CLI Implementation

The CLI is a dockable window.

To view it:

1. On the **View** menu, click **Debug Windows.**
2. Select **Command Line**, or

   Click the  button on the **View** toolbar.




This makes the **Command Line** window visible. To hide the **Command line** window, clear the **Command Line** check box.

## 2.11.6 About Command Scripts

### Creation

The Workbench supports the creation of command scripts for frequent execution of a set of Transactor or hardware commands. Command scripts are numbered 1 through 10.

To create a command script:

1. On the **Tools** menu, click **Customize**.

   The **Customize** dialog box appears.

2. Click the **Command Scripts** tab.
3. From the list on the left, select the command script number that you want assigned to the script.
4. In the box on the right, enter the Transactor commands you wish to have executed, just as you would enter them on the Transactor command line.
5. In the **Script name** box, enter the name you want associated with the command script. This name will be displayed in the tool tip and fly-by text when you position the mouse cursor over the corresponding command script toolbar button.
6. Click **Assign**. This assigns the commands and name to the selected command script number.
7. Repeat steps 3 through 6 for as many command scripts as you wish to assign, up to a total of 10.
8. Click **OK**.

Each command script (1-10) has an associated debug toolbar button  . To place a command script button in a toolbar, see Section 2.3.3.4.

#### Execution

Script files provide batch-style control of the Transactor or hardware.

To execute a script file:

1. Click the script file from the **Script Files** list in **FileView**.
2. On the **Debug** menu, click **Execute Selected Script**,

Or:

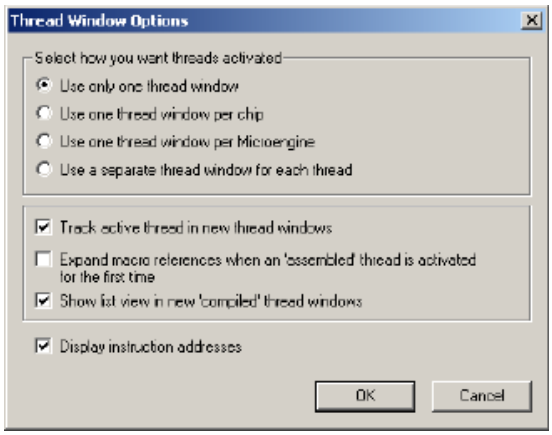
1. Right-click a script file in **FileView**.
2. Click **Execute Script** on the shortcut menu.

## 2.11.7 About Thread Windows

Thread windows differ from normal document windows in that they have a toolbar across the top (see Figure 2-6 and Figure 2-7). Setting and clearing breakpoints (see Section 2.11.10), displaying register contents (see Section 2.11.11), and viewing the instruction(s) currently being executed (see Section 2.11.7.10) are all done in thread windows. They cannot be performed in the source file windows.

### 2.11.7.1 Controlling Thread Window Activation

You can control how the thread windows are activated using the **Thread Window Options** dialog box.




To do this:

1. On the **Debug** menu**,** click **Thread Window Options**, or

   Click the  button in the toolbar of an open thread window.

   The **Thread Window Options** dialog box appears.

2. Select how you want threads activated. Select one of the following:

   a. **Use only one thread window**.
   Always reuse the currently open thread window and bring it to the top.

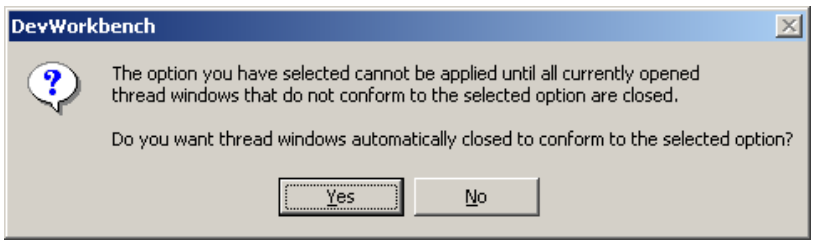
   b. **Use one thread window per chip**. Reuse a currently open thread window only if it displays a thread in the same chip as the thread being activated.

   c. **Use one thread window per Microengine**. Reuse a currently open thread window only if it displays a thread in the same Microengine as the thread being activated.

   d. **Use a separate thread window for each thread**. Always open a new thread window unless one is already open for the thread being activated, in which case the open window is brought to the top.

Whether or not you can select an option depends on the current thread window configuration. For example:

- If you have one or no thread window open, then all options are allowed.
- If you have two thread windows open, you cannot select option (a).
- If you have two or more thread windows open for different Microengines in the same chip, you cannot select option (a) or (b).
- If you have two or more thread windows open for different threads in the same Microengine, you cannot select option (a), (b) or (c).
- If you select an invalid option and click OK, the following message box appears:




  — If you click **No** the option reverts to the previous selection.

  — If you click **Yes** the Workbench closes the appropriate thread windows.

The activation option you select determines the behavior of the thread-selection toolbar.

- If you select option (d), all list boxes are disabled.
- If you select option (c), the chip and Microengine list boxes are disabled, allowing you to select a different thread.
- If you select option (b), the chip list box is disabled, allowing you to select a different Microengine and thread.
- If you select option (a), all list boxes are selected, allowing you to select a different chip, Microengine and thread.
- If the open project has only one chip, the chip list box is hidden in order to save toolbar space.

In the next area of the **Thread Window Options** dialog box:

3. Select **Track active thread in new thread windows** if desired (not available if you selected to view each thread in its own window).
4. Select **Expand macro references when an 'assembled' thread is activated for the first time** if desired.
5. Select **Show list view in new 'compiled' thread windows** if desired.
6. Select **Display instruction addresses** if desired (in list view only).
7. Click **OK** when done.

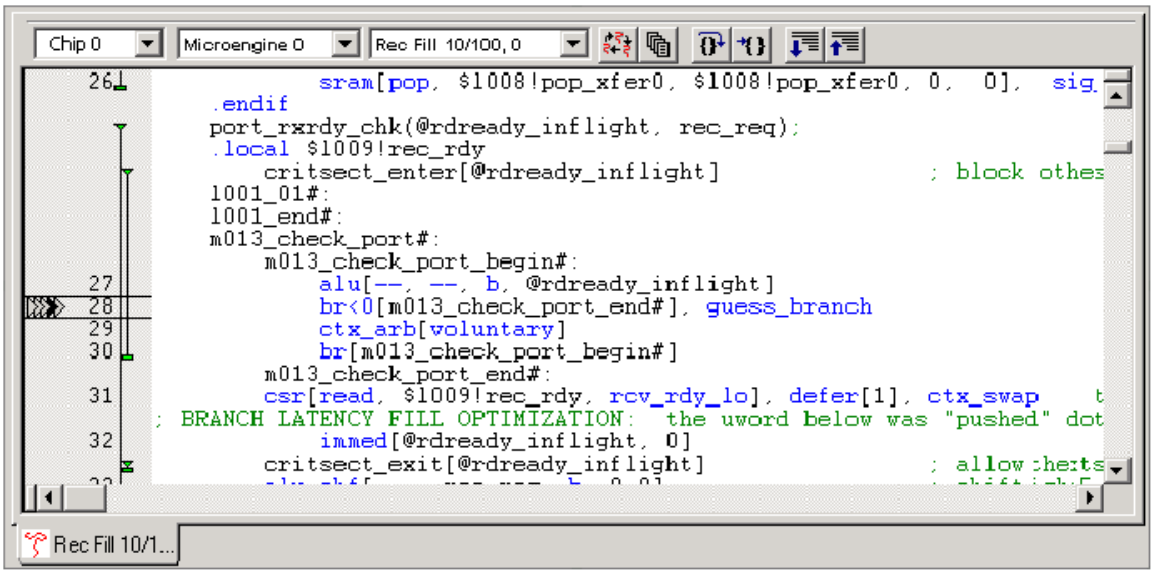
## 2.11.7.2 About Thread Window Controls

### Assembled Thread Windows

If a thread is in a Microengine whose list file is generated by the Assembler, then its thread window displays a 'list' view. This represents *flattened* code for the entire microstore, as contained in the .list file.

***Note:*** Setting and clearing breakpoints (see Section 2.11.10), displaying register contents (see Section 2.11.11), and viewing the instruction(s) currently being executed are also done in assembled thread windows. They cannot be performed in the source file windows.

**Figure 2-4. The Assembler Thread Window**




When a thread is being displayed in an assembled thread window, the toolbar contains the following controls:

**Chip** list — This control is not visible if your project has only one chip.

**Microengine** list — This control is disabled if you are using one thread window per Microengine or you are using a separate thread window for each thread.

**Thread** list — This control is disabled if you are using a separate thread window for each thread.

**Source** list (if in source view).
Here you can view any of the *.c files used to generate the .list file. When Microengine execution starts then stops, the Workbench changes the displayed source file to the one that generated the current instruction.

The toolbar contains following buttons:

 **Track Active Thread.** See Section 2.11.7.3.

 Display the **Thread Window Options** dialog box. See Section 2.11.7.1.

 **Step Over**. See Section 2.11.8.3.

 **Run to Cursor**. See Section 2.11.7.5.

 **Expand macros**. See Section 2.11.7.7.

 **Collapse macros**. See Section 2.11.7.7.

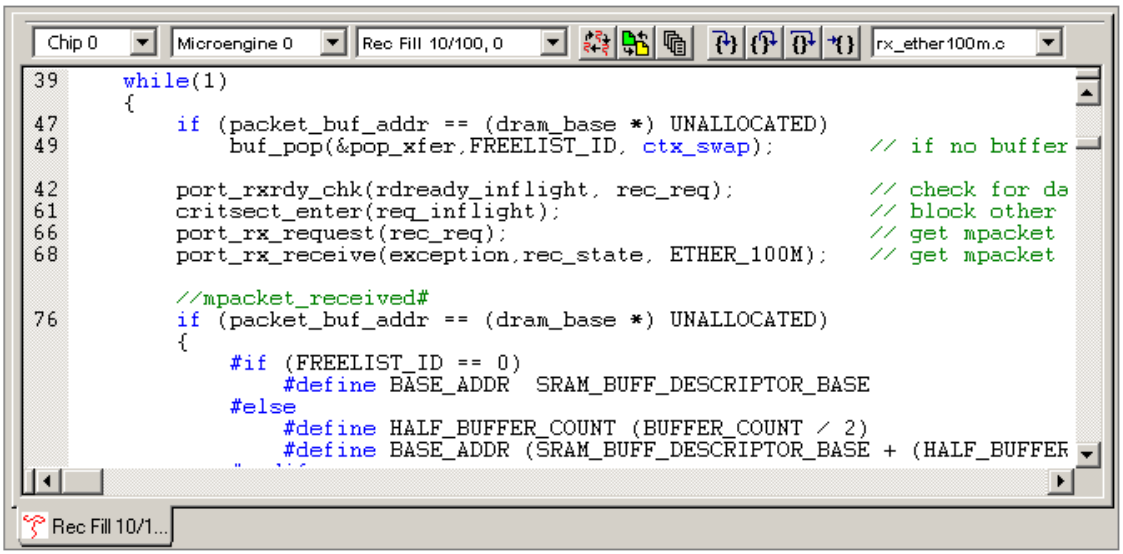
***Note:*** Not all buttons are available in hardware mode.

### Compiled Thread Windows

Compiler thread windows look the same as Assembler thread windows but have some differences. Display options are similar (see Section 2.11.7.1).

***Note:*** Setting and clearing breakpoints (see Section 2.11.10), displaying register contents (see Section 2.11.11), and viewing the instruction(s) currently being executed are also done in Compiler thread windows. They cannot be performed in the source file windows.

**Figure 2-5. The Compiler Thread Window**




When a thread is being displayed in a compiled thread window, the toolbar contains the following controls:

**Chip** list. This control is not visible if your project has only one chip.

**Microengine** list. This control is disabled if you are using one thread window per Microengine or you are using a separate thread window for each thread.

**Thread** list. This control is disabled if you are using a separate thread window for each thread.

**Source** list (if in source view).
Here you can view any of the *.c files used to generate the .list file. When Microengine execution starts then stops, the Workbench changes the displayed source file to the one that generated the current instruction.

The toolbar contains following buttons:


**Track the active thread.** See Section 2.11.7.3.


**Toggle View.** See Section 2.11.7.4.


Display the **Thread Window Options** dialog box. See Section 2.11.7.1.

| | | |
|---|---|---|
|  | **Step Into.** | See Section 2.11.8.4. |
|  | **Step Out.** | See Section 2.11.8.5. |
|  | **Step Over.** | See Section 2.11.8.3. |
|  | **Run to Cursor.** | See Section 2.11.7.5. |

*Note:* Unlike the assembled thread window, you cannot expand or collapse the display in a compiled thread window in list view.

*Note:* Not all buttons are available in hardware mode.

### 2.11.7.3 Tracking the Active Thread

You can specify that you want tracking of the active thread by clicking the  button in the thread window toolbar. This feature is available only if you have specified that you want only one thread window or one thread window per chip or per Microengine.

When Microengine execution is started then stopped and a different thread in the same Microengine becomes active, the thread window is automatically changed to display the active thread.

In the **Thread Windows Options** dialog box, specify whether new thread windows are opened with active thread tracking enabled by selecting or clearing **Track active thread** in new thread windows.

### 2.11.7.4 Toggling Views (compiled threads only)

You can toggle between source view and list view by clicking the  button. When switching from list view to source view, the Workbench displays the source file that contains the line that generated the list view line where the cursor is located. The generating source line is scrolled to be visible in the list view. When switching from source view to list view, the Workbench scrolls to the first list view line generated by the source line where the insertion cursor is located.

### 2.11.7.5 Running to Cursor

If you are debugging in Simulation mode, you can place the cursor at a point in the code and then you can Run to Cursor.

To do this:

1. Place the cursor on a line in a thread window by clicking in that line.
2. On the **Debug** menu, click **Run Control**, then click **Run To Cursor**, or

   Click the  button in the Thread window.

Or:

Right-click in the line to which you want to run, then select **Run To Cursor** from the shortcut menu.

If the line is in the source view of a compiled thread or if it contains a collapsed macro reference in an assembled thread, then the simulation runs until the first generated instruction is reached.

**Run to Cursor** can be performed only on lines that generated instructions.

### 2.11.7.6 Activating Thread Windows

Once microcode is loaded, you can directly access the execution state of all the threads in the project.

To explicitly activate a thread window, do this:

1. Double-click the thread name in the **ThreadView** or the **Thread Status** window, or

   Right-click the desired thread in the **ThreadView** or the **Thread Status** window and click **Open Thread Window**, or

   Change the selection(s) in the thread-selection toolbar in the thread window.

The thread window is implicitly activated by:

- Stopping at a breakpoint. The thread in which this occurred is activated.
- Selecting **Go To Instruction** from the shortcut menu in either the thread history or queue status window. The thread in which the instruction was executed is activated.

When Microengine execution stops for any reason other than a location breakpoint—such as, a break-on-change occurs, or you click Stop, or a packet count limit was reached by the IX bus device simulation—the Workbench determines the active thread for each Microengine and does one of the following:

- If the active thread is already activated in a thread window, the window is simply scrolled to display the current instruction or source line.
- If the active thread isn't already activated and there is a thread window in which a different thread in the same Microengine is activated, then the active thread is activated in that window if you have specified that you want tracking of the active thread. (A toolbar button in the thread window allows you to enable or disable this feature.)
- If the active thread isn't already activated and there are no thread windows in which a different thread in the same Microengine is activated, then the active thread is not activated.

#### Thread Window Title Bar


The title displayed on the thread window shows the thread name and Microengine numbers. Also displayed is the currently executing PC and whether the thread is active or swapped out.

#### Thread Window Contents

A thread window displays the output of the Assembler as opposed to original source code. The Assembler output differs from source code in that:

- Symbols are replaced with actual values.
- Instructions may be reordered due to optimization.
- Names of local register are adorned by a prefix, etc.
- If you built a Microengine image from multiple source files by using the #include directive, then the associated thread window displays the modified output from the combined sources.

## 2.11.7.7 Displaying, Expanding, and Collapsing Macros (assembled threads only)

By default, all macros are collapsed. A green triangle to the left of the instruction indicates that the instruction is a fully collapsed macro (see Figure 2-6).

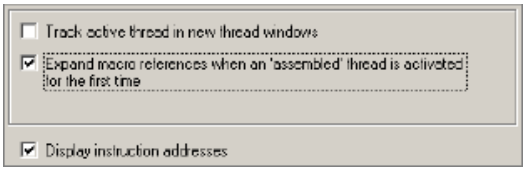
### First time thread activation

You can specify whether macro references are fully expanded or collapsed when an assembled thread is activated for the first time. To do this:

1. On the **Debug** menu, click **Thread Window Options**.

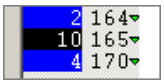
   The **Thread Window Options** dialog box appears.

2. Enable or clear the **Expand macro references when an 'assembled' thread is opened for the first time**.

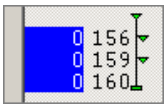



When you re-activate an assembled thread, the Workbench restores the state of macro expansion that existed when the thread was deactivated. However, when you stop debugging, the macro expansion state is no longer remembered.

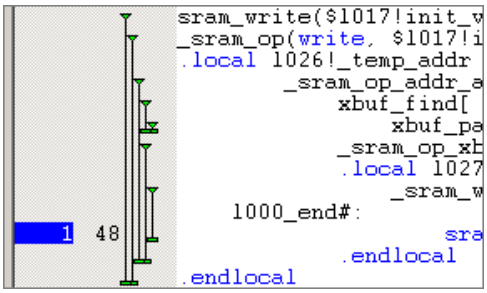
**Figure 2-6. Expanding Macros**




Collapsed




Expanded One Level




Fully Expanded

### Macro Marker Display

If you don't see the macro markers you may have to enable them.

To display macro markers:

1. Right-click the thread window.
2. Click **Display Macro Markers** on the shortcut menu.

### Macro Expansion

To expand a collapsed macro:

1. Right-click on the triangle or anywhere on the instruction line.
2. Click **Expand Macro One Level**, or

   Click **Expand Macro Fully**.

#### Macro Collapse

You can only fully collapse an expanded macro, not one level at a time. To do this:

1. Right-click anywhere on the instruction line.
2. Click **Collapse Macro**.

#### Expand and Collapse of All Macros at Once (Assembled Threads Only)

You can expand and collapse all macros at the same time. Do the following:

- To expand all macros one level, click the  button.
- To collapse all macros one level, click the  button.

Note that this is the only way to collapse a macro one level at a time.

#### Go To Source

To go to the source line corresponding to a line in a thread window:

1. Place the insertion cursor on the line.
2. On the **Debug** menu, click **Go To Source**.

   The Workbench:

   — Opens a document window with the source file.

   — Places the insertion cursor at the beginning of the requested line.

   — Scrolls the line into view.

You can also right-click the thread window line and click **Go To Source** from the shortcut menu.

### 2.11.7.8 Displaying and Hiding Instruction Addresses

To display or hide the microstore address at which each instruction in a thread window is located:

1. Right-click in the thread window.
2. Select or clear **Display Instruction Addresses** on the shortcut menu.

This toggles displaying of the addresses of the microstore instructions. You can also do this using the **Thread Window Options** dialog box.

If macro references are expanded, instruction addresses are displayed on the generated instruction lines. If references are collapsed, addresses are displayed on the macro reference lines, with the address being that of the first instruction generated by that reference.

***Note:*** The displaying of instruction addresses affects all thread windows and is saved as a global option that is in effect across all projects.

### 2.11.7.9 About Instruction Markers

During a typical debugging session, thread windows display several types of instruction markers. An instruction marker displays on the left side of the thread window, in the same location as bookmarks and breakpoints. The types are summarized in Table 2-2:

**Table 2-2. Instruction Markers**

| Marker Name | Symbol | Function |
|---|---|---|
| Swapped Out |  | Marks the instruction to be executed when the thread's context is swapped back in. |
| History |  | Marks the instruction that was executing in pipe stage 3 at the cycle associated with the thread history window's cycle marker. |
| Pipe Stage |  | Marks the instructions executing in each of the 5 pipeline stages. |

#### Assembled Thread

In an assembled thread, if line containing the current instruction is displayed, then the instruction marker points to it. If the line is hidden because it is in a collapsed macro, then the instruction marker points to the line containing the collapsed macro.

#### Compiled Thread

In the list view for a compiled thread, the instruction marker points to the current instruction. In the source view for a compiled thread, the instruction marker points to the C source line that generates the current instruction.

### 2.11.7.10 Viewing Instruction Execution in the Thread Window

During a simulation session, the Pipe Stage marker allows you to view which instruction is inside each of the five stages of the pipeline. This marker contains up to five stacked arrowheads that correspond to each of the five pipeline stages. The leftmost arrowhead represents stage 0, and the rightmost arrowhead represents stage 4. The default is stage 3.

#### Colors

The arrowheads are color-filled according to the state of the instruction in the pipeline stage. By default, the Workbench uses:

**Black** for instruction executing.

**Yellow** for instruction aborted.

**Red** for thread stalled.

If a thread has a different instruction in each of the pipe stages, then the thread window will have five Pipe Stage markers, one on each of the five instructions. Each marker will have a different arrowhead filled with the appropriate color. For example, if an instruction is executing in stage 3, then its marker will have the stage 3 arrowhead filled with black with all other arrowheads unfilled. If an instruction is aborting in stage 2, then its marker will have the stage 2 arrowhead filled with yellow with all other arrowheads unfilled. If an arrowhead is not color-filled, it means that the instruction that the marker points to is not in the corresponding pipeline stage.

#### Same Instruction in More Than One Pipeline Stage

It is possible, due to branching, for the same instruction to be in more than one pipeline stage. In this case, the Pipe Stage marker on that instruction will have multiple arrowheads filled in, possibly with different colors. This also means that there will be fewer than five markers in the thread window.

### Context Swapping Issues

When a context swap is in progress, the latter stages of the pipeline have instructions from the context being swapped out and the early stages have instructions from the context being swapped in. In this case, the thread windows for both contexts have Pipe Stage markers displayed. However, the marker for each thread window will show arrowheads only for those stages in which the thread has instructions.

For example, if a thread only has an instruction in stage 4, then its marker will only show a single arrowhead, corresponding to stage 4. The other thread marker will show arrowheads for each of the four stages in which it has instructions.

When a context is completely swapped out, its thread window displays all five arrowheads unfilled to mark the instruction at which execution will resume when the context is swapped back in.

## 2.11.8 About Run Control—Simulation Mode

Run Control lets you govern execution of the Microengines. Different control operations are available from the Workbench depending on whether you are in Simulation or Hardware mode mode (see Table 2-1).

In Simulation mode, the Workbench provides the controls for running microcode in a dockable Run Control window.

To display the **Run Control** window:

1. On the **View** menu, click **Debug Windows**.
2. Select or clear **Run Control** to toggle visibility of the **Run Control** window, or

   Click the  button on the **View** toolbar.

### 2.11.8.1 Single Stepping

Single stepping has four variations:

| | |
|---|---|
| **Microengine Step** | Performed on Microengines (see Section 2.11.8.2). |
| **Step Into** | Performed on a single thread in a compiled thread window only (see Section 2.11.8.4). |
| **Step Over** | Performed on one thread (see Section 2.11.8.3). |
| **Step Out** | Performed on a single thread in a compiled thread window only (see Section 2.11.8.5). |

### 2.11.8.2 Stepping Microengines

To single step one cycle:

1. Click **Step** in the **Run Control** window, or

   On the **Debug** menu**,** click **Run Control,** then click **Step Microengines**, or

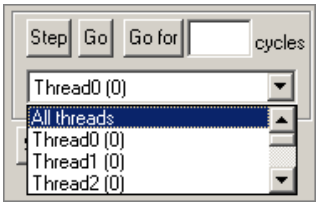
   Press SHIFT+F10, or

   Click the  button.

#### All Microengines

To single step all Microengines, regardless of which threads are running:

- Select the **All threads** entry from the list under the **Step** button in the **Run Control** window.




#### A Specific Thread

To single step one cycle of a specific thread:

- In the **Run Control** window, select the thread's entry from the list under the **Step** button.

### 2.11.8.3 Stepping Over

Step Over allows you to execute as many machine cycles as it takes complete the current line in the thread window. To Step Over:

- On the **Debug** menu**,** click **Run Control,** then click **Step Over**, or

  Click the  button, or

  Right-click in the thread window and click **Step Over** from the shortcut menu, or

  Press F10.

### 2.11.8.4 Stepping Into (compiled threads only)

**Step Into** executes as many Microengine cycles as it takes to execute the current line in the thread window, whether it is a microinstruction line in a list view or a C source line in a source view. Stepping into is supported only for compiled threads.

To Step Into do the following:

- On the **Debug** menu, click **Run Control,** then click **Step Into**, or

  Click the  button in the thread window's toolbar, or

  Right-click in the thread window and select **Step Into** from the shortcut menu.

### 2.11.8.5 Stepping Out (compiled threads only)

**Step Out** executes as many Microengine cycles as it takes to complete the thread's current function and return to the calling function. Stepping out is supported only for compiled threads.

To step out, do the following:

- On the **Debug** menu, click **Run Control,** then click **Step Out**, or

  Click the  button in the thread window's toolbar, or

  Right-click in the thread window and select **Step Out** from the shortcut menu.

### 2.11.8.6 Executing Multiple Cycles

#### All Microengines

To run for a specified number of cycles in all Microengines, regardless of which threads are running:


1. Select **All threads** in the list under the **Go for** button.
2. Type the number of cycles in the box to the right of the **Go for** button.
3. Click **Go for**.

All Microengines run until the specified thread has executed the specified number of cycles.

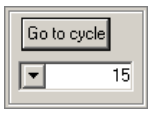
#### A Specific Thread

To run for a specified number of cycles in a specific thread:

1. Select the thread's entry from the list under the **Go for** button.
2. Type the number of cycles in the box to the right of the **Go for** button.
3. Click **Go for**.

### 2.11.8.7 Running to a Specific Cycle

To run until a specified cycle count is reached:




1. Type the cycle count in the box under the **Go to cycle** button.
2. Click **Go to cycle**.

### 2.11.8.8 Running to a Label or Microword Address

To run until a specific microcode label or microword address is reached by a thread:

1. Enter the label(s) and/or address(es) into the appropriate boxes.
2. Click **Go to label/address**.

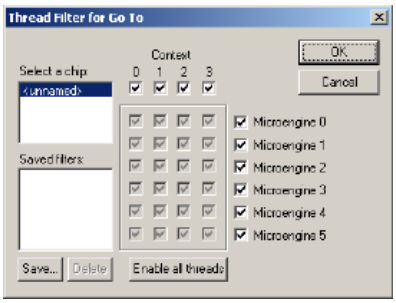
#### Thread Specification

To specify the thread(s) for which an operation is effective:

1. Click **Set thread filter**.

   The **Thread Filter to Go To** dialog box appears.

   The dialog box to the right shows all threads selected (equivalent to clicking Enable all Threads).




2. Select a chip (if more than one); select a thread filter (if available); save the filter you just created; delete a saved filter(s); and select a previously saved filter.
3. You can also enable any combination of threads by selecting or clearing **Microengine** and **Context**.
4. Click **OK** when done to set the filter.

5. Click **Go to label/address** to run.

Execution stops when any one of the selected threads reaches the target label or microword.

### 2.11.8.9 Running Indefinitely

To run the microcode indefinitely:

1. On the **Debug** menu, click **Run Control**, then click **Go**, or
   Click the Go button in the **Run Control** window, or
   Click the  button on the **Debug** toolbar, or
   Press F5.

Microcode execution stops only if a breakpoint is reached or if you manually stop execution (see Section 2.11.8.10).

### 2.11.8.10 Stopping Execution

To stop microcode execution at any time:

1. On the **Debug** menu, click **Run Control**, then click **Stop**, or
   Click Stop in the **Run Control** window, or
   Click the  button on the **Debug** menu, or
   Press SHIFT+F5.

### 2.11.8.11 Resetting the Simulation

Reset executes a Transactor sim_reset command, which puts the simulation back to the state after the original initialization was done. After the reset, the Workbench re-executes the options specified by the **Startup** page of the **Simulation Options** dialog box. However, if you specified that the Workbench should initialize the model, it doesn't repeat the chip and memory definition commands and the init command since they are unnecessary.

To reset the simulation:

1. On the **Debug** menu, click **Run Control**, then click **Reset**, or
   Click the  button on the **Debug** toolbar, or
   Press CTRL+SHIFT+F12.

## 2.11.9 About Run Control—Hardware Mode

Run Control in Hardware mode is limited to the following commands:

| | |
|---|---|
| **Hop** | See Section 2.11.9.1. |
| **Run indefinitely**. | See Section 2.11.9.2. |
| **Stop** | See Section 2.11.9.3. |
| **Reset** | See Section 2.11.9.4. |

The dockable **Run Control** window is not available in Hardware mode. Run Control commands are issued using the **Debug** toolbar or from the **Debug** menu. The **Debug** toolbar appears when you enter debug mode if it is not already visible.


### 2.11.9.1 About Hopping

A Hop lets each Microengine run until a context switch occurs, at which time the Microengines are placed in the Paused State.

#### Invocation

A Hop is invoked using the Hop command. A Hop can be invoked only if all of the Microengines are in a Stopped or Paused State.

#### Completion

The Hop operation completes when all of the Microengines have executed a context swap and are paused. A context swap occurs as a result of a ctx_arb microcode command or a ctx_swap optional token on a microcode command.

#### Hop and Tight Loop Problem

If a thread executes a tight loop, such as waiting for an inter-thread signal, the thread may not release its context. If one or more Microengines does not swap its context, the Hop operation fails and you see a message similar to, "`Microengine 0 failed to pause`." If this occurs, you are not able to execute another Hop until all of the Microengines are either stopped or reset.

#### Hop While Microengine is in the Run State

If you do invoke a Hop operation while one or more Microengines is in the Running State, you will see the message, "Hop MicroEngines operation could not be performed because MicroEngine is active".

#### Hop Failures

If a Hop fails, you can determine which Microengines are active by viewing the **Thread Status** window from the Workbench. An active Microengine has a black arrow next to the thread ID that is running. A paused Microengine will have a gray arrow next to the thread ID that is scheduled to resume execution.

### 2.11.9.2 Running Indefinitely

To run the microcode indefinitely:

- Click the go  button on the **Debug** toolbar, or

  On the **Debug** menu, click **Run Control** then select **Go**, or

  Press F5.

Microcode execution stops only if a breakpoint is reached or if you manually stop execution.

### 2.11.9.3 Stopping Execution

To stop microcode execution at any time:

- Click the stop button on the **Debug** toolbar, or

  Click **Debug, Run Control, Stop**, or

  Press SHIFT+F5.

#### Failure to Stop Problem

When you issue the Stop command, the debug library on the StrongARM core waits for a context swap in each of the Microengines before completing the command. If the active thread on one or more Microengines fails to swap, you see the message, "`Microengine 0 failed to pause.`" You can determine which threads are still active by viewing the **Thread Status** window.

### 2.11.9.4 Resetting the Hardware

You can reset the microcode and hardware by issuing the Reset command, which places each Microengine in the Reset State.

#### Simulation Mode

When the Reset command is issued in Simulation mode, each Microengine marks its first thread as ready to run next.

#### Hardware Mode

In Hardware mode, the thread that was marked as ready to run at the time of the reset is the one to run when execution resumes. This may cause the results of a Hop operation or a breakpoint to vary each time the hardware is reset. You can determine which thread is marked as ready to run by viewing the **Thread Status** window.

To perform a Reset:

- On the **Debug** menu, click **Run Control,** then click **Reset**, or

  Click the button on the **Debug** toolbar, or

  Press CTRL+SHIFT+F12.

## 2.11.10 About Breakpoints

In the source view, when you set a breakpoint on a line, a breakpoint marker is displayed on that line and a breakpoint is set on the first instruction that it generates. In the list view, it is possible to set breakpoints on multiple lines that are generated by the same C source line. In this case, the marker that is displayed on the source line in the source view depends on the state of the breakpoints on the generated lines.

- If all breakpoints have the same status, then the source line marker reflects that status. In this situation, you can perform a breakpoint action on the source line and the action is performed on all the breakpoints on the generated lines. For example, if all are disabled, then the disabled-breakpoint marker is displayed on the source line. If you enable the breakpoint on the source line, all breakpoints on generated lines are enabled.
- If the breakpoints do not have the same status, then a distinct marker consisting of a red circle filled with dark gray is displayed on the source line. In this situation, the only supported action is to enable all the breakpoints by executing an Insert/Remove Breakpoint command.

In an assembled thread, when you set a breakpoint on a line containing a collapsed macro reference, a breakpoint marker is displayed on that line and a breakpoint is set on the first instruction that the macro generates. If you then expand that macro reference, the breakpoint marker is displayed on the generated line.

For situations where there are breakpoints on multiple lines generated by a collapsed macro, the breakpoint marker and supported actions are the same as described above for the C source line.

#### Thread Window Action Taken

When a breakpoint is reached during execution:

- The thread window that reached the breakpoint is activated.
- The appropriate instruction is displayed and marked.
- A message box appears (if set) indicating the breakpoint was reached.

  To disable the display of this message box, on the **Debug** menu, select or clear the **Report Breakpoint Hit** option.

#### Conditional And Unconditional

A breakpoint can be unconditional or conditional.

- An unconditional breakpoint on an instruction halts execution when any context in the Microengine reaches that instruction.
- A conditional breakpoint halts execution only when one of the specified contexts reaches that instruction.

### 2.11.10.1 Setting Breakpoints in Hardware Mode

#### Restrictions

You can set breakpoints in Hardware mode with the following restrictions:

- Each breakpoint you insert causes the Debug library in the StrongARM core to place a breakpoint routine in unused Control Store space within the Microengines. Consequently, the number of breakpoints you can insert will be limited by the size of your microcode image.
- You may be prevented from setting a breakpoint on certain instructions because processing the breakpoint will adversely affect the thread's execution state or register contents.

  If this occurs, you see the following message when you attempt to insert the breakpoint: "Breakpoint can't be set at line xx because of Microcode sequencing restrictions".

If breakpoints are set in multiple Microengines, it is possible to hit more than one breakpoint before all of the Microengines have paused.

As with Hop processing, if a thread running on a Microengine does not release its context, it will continue to run. You should check the **Thread Status** window after a breakpoint has been reached to determine if any threads are still active.

### 2.11.10.2 About Breakpoint Markers

When a breakpoint is set on a microword address, the Workbench displays a breakpoint marker in the left-hand margin of the corresponding line in the thread window. The marker's appearance depends on the properties of the breakpoint.

The different markers and their meanings are described below:

| | |
|---|---|
| Unconditional breakpoint | Enabled in all threads in Microengine. |
| Unconditional breakpoint | Disabled in all threads in Microengine. |
| Conditional breakpoint | Set and enabled in this thread but not set in all threads in the Microengine. |
| Conditional breakpoint | Set but disabled in this thread but not set in all threads in the Microengine. |
| Conditional breakpoint | Not set in this thread but set and enabled in one or more other threads in the Microengine. |
| Conditional breakpoint | Not set in this thread but set and disabled in one or more other threads in the Microengine. |
| Special breakpoint | The states of two or more breakpoints in the generated code are different, so the corresponding line in the source code gets a special breakpoint marker. In this situation, the only supported action is to enable all the breakpoints by executing and **Insert/Remove Breakpoint** command. |

### 2.11.10.3 Inserting and Removing Breakpoints

To insert a breakpoint in a Microengine:

1. Open a thread window for one of the threads in the Microengine.
2. Place the insertion cursor on the line where you wish to insert the breakpoint.
3. On the **Debug** menu, click **Breakpoint**, then click **Insert/Remove**, or
   Click the  button on the **Debug** toolbar, or
   Press F9.

Or:

1. Right-click the line at which you wish to insert the breakpoint.
2. Click **Insert/Remove Breakpoint** from the shortcut menu.

#### Conditional Breakpoints

By default, the breakpoint is inserted as unconditional. To make the breakpoint conditional you must change its properties (see Section 2.11.10.4).

#### Hardware Mode Restrictions

When debugging in Hardware mode, you cannot set breakpoints on instructions that:

- Are in defer shadows, or
- Are indirect branches.

#### Breakpoint Removal

To remove a breakpoint in a Microengine:

1. Open a thread window for one of the threads in the Microengine in which the breakpoint is set.
2. Place the insertion cursor on the line at which you wish to remove the breakpoint.
3. On the **Debug** menu, click **Breakpoint**, then click **Insert/Remove**, or

   Click the  button on the **Debug** toolbar.

   Press F9.

   or

1. Right-click the line at which you wish to remove the breakpoint.
2. Click **Insert/Remove Breakpoint** from the shortcut menu.

The breakpoint is removed in all contexts, regardless of whether it was conditional or unconditional.

#### Removal of All Breakpoints

To remove all breakpoints in all Microengines:

- On the **Debug** menu, click **Breakpoint**, then click **Remove All**, or

  Click the  button on the **Debug** toolbar.

### 2.11.10.4 Enabling and Disabling Breakpoints

To enable or disable breakpoints on code locations, do the following:

1. Place the insertion cursor on the line at which you wish to enable/disable a breakpoint.
2. On the **Debug** menu, click **Breakpoint**, then click **Enable/Disable**, or

   Click the  button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To put this button there, see Section 2.3.3.4.), or

   Press CNTRL+F9.

Or:

1. Right-click the line at which you wish to enable/disable a breakpoint.
2. Click **Enable/Disable Breakpoint** from the shortcut menu.

To disable all breakpoints:

- On the **Debug** menu, click **Breakpoint**, then click **Disable All**, or

  Click the  button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To put this button there, see Section 2.3.3.4.).

To enable all breakpoints:

- On the **Debug** menu, click **Breakpoint**, then click **Enable All**, or

  Click the  button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To put this button there, see Section 2.3.3.4.).

### 2.11.10.5 Changing Breakpoint Properties

To change the breakpoint properties,

1. Open a thread window for one of the threads in the Microengine where the breakpoint is set.
2. Place the insertion cursor on the line where you want to change breakpoint properties.
3. On the **Debug** menu, click **Breakpoint**, then click **Properties**.

   The **Breakpoint Properties** dialog box appears. The dialog box shows the chip, Microengine, microword address, and thread window line number where the breakpoint is set.

In this dialog box you can:

- Select or clear **Enabled** to enable or disable the breakpoint.
- Click **Unconditional** to make the breakpoint unconditional.
- Click **Conditional** to make the breakpoint conditional (that is, it applies to selected contexts in the Microengine)

  Select the boxes for the contexts for which you want to apply the breakpoint.
- Click **Remove** in the **Breakpoint Properties** dialog box to remove the breakpoint.

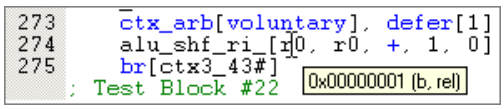
## 2.11.11 Displaying Register Contents

When program execution is stopped, you can display register contents directly from instruction context in a thread window.

To do this:

1. Position the cursor over the register symbol.
2. Wait for a moment.

   The Workbench displays the contents of the register assigned to that symbol as a pop-up window beneath the cursor.




### Hex or Decimal Display

To control whether the data is displayed in decimal or hexadecimal format:

1. Right-click in the thread window.
2. Select or clear **Hexadecimal Data** on the shortcut menu.

## 2.11.12 About Data Watch

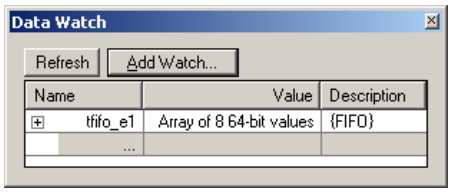
In debug mode, you can monitor the values of simulation states using the **Data Watch** window.

To do this:

- On the **View** menu, click **Debug Windows**, then click **Data Watch**, or

  Click the  button on the **View** toolbar.

This toggles visibility of the Data Watch window. The window contains a list with three columns:




| | |
|---|---|
| **Name** | Contains the name of the state being watched. |
| **Value** | Displays the state's value. |
| **Description** | Contains information such as which chip, Microengine or thread the watch pertains to. |

#### Values Updates

Watch values are updated whenever microcode execution stops. To force updating the values at other times, click **Refresh**.

### 2.11.12.1 Data Watches in C Thread Windows

In C thread windows, data watches can be set for C variables by right-clicking on the variable in the thread window and selecting **Set Data Watch for:<variable_name>**.

- When the variable is in scope its value appears in the **Data Watch** window.
- When the variable is out of scope, the phrase out of scope appears in the **Value** field for the watched variable.
- Variables that contain C structures are displayed hierarchically, with the member variables displayed on separate lines in the watch window. You can expand and collapse the display of the member variables.

*Note:* Not all variables can have data watches set. Many variables are optimized away by the Compiler and the Compiler does not provide any debug data for those variables. The workbench doesn't know that the text you select is a variable if it doesn't have any debug data. If this is the case, the **Set Data Watch** option on the shortcut menu is unavailable.

### 2.11.12.2 Entering a New Data Watch

To enter a new data watch:

1. Right-click anywhere in the **Data Watch** window.
2. Click **New Watch** from the shortcut menu, or

   Double-click the blank entry at the bottom of the data watch list.
3. Type the name of the state you want to watch.

#### Array States

For states that are arrays, such as SDRAM, you must enter a range of array locations to be watched. The format for the range is the same as that for the Transactor:

| | |
|---|---|
| **[m]** | to watch a single location of an array. For example, sram[1] watches location one in SRAM. |
| **[m:n]** | to watch locations m through n inclusive. For example, sdram[0:3] watches locations 0 through 3 in SDRAM. And since a data watch range can be specified in ascending or descending order, you could specify this watch as sdram[3:0]. |
| **[m:+n]** | to watch location m plus the n locations following it. For example, sdram [5:+3] watches locations 5 through 8. |

### FIFO Elements

An element that is a FIFO can be watched either as an array or a FIFO.

- Watching such an element as an array shows the locations in physical order.
- Watching it as a FIFO shows the locations in FIFO order.

To watch as a FIFO, use a "[[" instead of "[" to begin the range qualifier.

### Bit Range

You can specify a bit range to be watched. The format for a bit range is:

**<m>** to watch only bit m of a state. For example, f0.ctl.p0_addr<12> watches only bit 12 of the stage 0 address.

**<m:n>** to watch bits m through n of a state, with m being greater than n. For example:

sdram[0:3]<12:10> watches bits 12 through 10 of sdram locations 0 through 3.

### Segments

Data watch values are broken into 32-bit segments. For example:

A 64-bit value displays as 0xcafecafe 0xcafecafe.

### Save

Data watches are saved with project debug settings.

**Table 2-3. Data Watch Values**

| Variable Name Inserted In the Name Column | Comments |
|---|---|
| sim.error_count | Watch a simulation state variable. Not available on real hardware. |
| fbi.scratch_pad[0:3] | Watch only the first four words of fbi.scratch_pad. |
| sram[0:10] | Watch an array of 11 values each 32 bits long. |
| sram[0:10]<3:0> | Watch an array of 11 values, only the first four bits. |
| sdram[20:22] | Watch an array of 3 values each 64 bits long. |

## 2.11.12.3 Watching Control and Status Registers and Pins

The Workbench recognizes the control and status register (CSR) and pin names described in the *IXP1200 Network Processor Programmer's Reference Manual*. For example, you can enter the name IREQ to watch the FBI Interrupt Register. If your project contains multiple chips, you are prompted to select which chip's register to watch. Similarly, if the register is Microengine-based, you are prompted to select which Microengine register to watch.

### Named Elements

To add a data watch by selecting a named element from a list:

1. Click **Add Watch** or right-click in the **Data Watch** window and click **Add Watch** from the shortcut menu.

   The **Add Data Watch** dialog box appears.

2. Click the category of named element that you want listed.

A list of recognized element names appears on the right.

3. In the list, select one or more elements you want to watch.

*Note:* Use the left mouse button in combination with the SHIFT or the CTRL keys to select multiple elements.

4. Click **Add Watch** to have a watch added for each selected element.

   If you select from the list of MicroEngine CSRs, you are prompted with a dialog box to select in which Microengine you want the watch to be done. Similarly, if you select from a list of chip-specific elements, such as the Memory CSRs, and your project contains multiple chips, you are prompted to select a chip.

5. When you are finished adding your watches, click **Close**.

*Note:* In Hardware mode, you cannot create a Data Watch for the Pins category as well as certain CSRs.

### 2.11.12.4 Watching General Purpose and Transfer Registers

To watch general purpose registers (GPRs) and transfer registers whose symbols are defined in your microcode:

1. Open the thread window containing the microcode.
2. Right-click the register name.
3. Click **Set Data Watch for**: from the shortcut menu.

You can also add a watch by selecting a register name from a list:

1. Click **Add Watch** or right-click in the **Data Watch** window and click **Add Watch** from the shortcut menu.

   The **Add Data Watch** dialog box appears.

2. Select the chip and Microengine in which you want to watch registers.
3. Select whether you want GPRs listed by selecting or clearing **List GPRs**.
4. Select whether you want transfer registers listed by selecting or clearing **List Transfer Regs**.

   Based on your selections, the relative registers are listed in the **Relative registers** list box and the absolute registers are listed in the **Absolute registers** list box.

5. Select one or more registers from either or both lists and click **Add Watch** to add watches for the selected registers.

   If you select relative registers, then you must specify which threads you want to watch by selecting or clearing the four check boxes beneath the relative registers list.

6. When you are finished adding your watches, click **Close**.

### 2.11.12.5 Deleting a Data Watch

To delete a data watch:

- Right-click the watch to be deleted in the **Data Watch** window, and click **Delete Watch** from the shortcut menu, or

  Select the watch to be deleted, then, on the **Debug** menu, click **Data Watch**, then click **Delete**, or

Select the watch to be deleted and press DELETE.

### 2.11.12.6 Changing a Data Watch

To change a data watch:

1. Right-click the watch to be changed in the **Data Watch** window and select **Edit Name** on the shortcut menu, or

   Select the data watch whose name you want to change, then, on the **Debug** menu, click **Data Watch**, then click **Edit Name**, or

   Double-click the name to be changed.
2. Type the state name.
3. Press ENTER.

### 2.11.12.7 Changing the Data Watch Radix

To select whether watch values are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **Data Watch** window.
2. Click **Hexadecimal Data** on the shortcut menu to display values in hexadecimal format or clear it to display in decimal format.

### 2.11.12.8 Depositing Data

To change the value of a simulation state in the **Data Watch** window:

1. Right-click the value to be changed and click **Edit Value** on the shortcut menu, or

   Select the data watch whose value you want to change, then, on the **Debug** menu, click **Data Watch**, then click **Edit Value**, or

   Double-click the value to be changed.
2. Type the new value in either hexadecimal or decimal format and press ENTER. Hexadecimal values must be preceded by a ‘0x’.

*Note:* In Hardware mode, you cannot change the contents of the FIFO elements and certain CSRs.

### 2.11.12.9 Breaking on Data Changes

In Simulation mode, you can halt microcode execution when a state’s value changes.

A red dot (the breakpoint symbol) appears before each state on which a break-on-change is set.

You can set and remove a break-on-change on aggregate state (an array or FIFO) or on its individual elements. Setting or removing a break-on-change on an aggregate state affects all its elements. If some but not all of an aggregate state’s elements have break-on-change set, then a half-filled breakpoint symbol is displayed on the aggregate state.

When the value changes for a state on which a break-on-change is set, microcode execution halts and a message box is displayed containing the state name along with its old and new values.

*Note:* Breaking on data changes is not supported in Hardware mode.

### Set

To break execution on a changed data value:

1. Right-click the name or value of the state and click **Set Break On Change** on the shortcut menu, or click the name or value of the state.
2. On the **Debug** menu, click **Data Watch**, then click **Set Break On Change**.

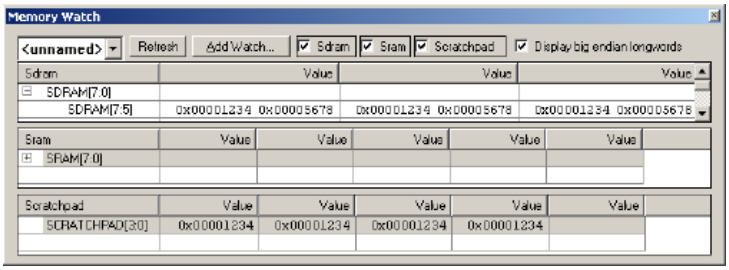
### Remove

To remove a break-on-change:

1. Right-click the name or value of the state and click **Remove Break On Change** on the shortcut menu, or click the name or value of the state.
2. On the **Debug** menu, click **Data Watch**, then click **Remove Break On Change**.

## 2.11.13 About Memory Watch

In debug mode, you can monitor the values of SDRAM, SRAM, and Scratchpad memory locations using the **Memory Watch** window.




### Visibility

To toggle visibility of the **Memory Watch** window:

- On the **View** menu, click **Debug Windows**, then click **Memory Watch**, or
  Click the  button on the **View** toolbar.

### Chip Selection

You select which chip's memory is watched using the list in the upper left corner of the **Memory Watch** window. Chip selection is synchronized with chip selection in the **History**, **Thread Status** and **Queue Status** windows.

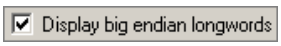

### Subwindows

The **Memory Watch** window comprises three subwindows, one for each memory type. Each memory type has a check box at the top of the **Memory Watch** window to control the visibility of the subwindow.


Each subwindow contains a multicolumn tree. The first column contains the range of the location(s) being watched, e.g. sdram[0:3]. The values of the locations are displayed in columns 1 through n. The number of value columns varies with the width of the **Memory Watch** window, with a minimum of one value column.

### Endianness

The **Display big endian longwords** check box enables SDRAM values to be displayed in big-endian or little-endian format.

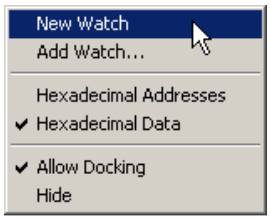



### Values Updates

Watch values are updated whenever microcode execution stops. To force an update of the values at other times, click **Refresh** at the top of the **Memory Watch** window.


## 2.11.13.1 Entering a New Memory Watch

To enter a new memory watch:




1. Right-click anywhere in the name or value column of the **Memory Watch** subwindow (either Sram. Sdram, or Scratchpad) and click **New Watch** on the shortcut menu, or

   Double-click the blank entry at the bottom of the data watch list for that subwindow.

2. Type the range of memory locations you would like to watch.

   The format for the range is the same as that for the Transactor:

   **[m]** To watch a single location. For example, sram[1] watches location one in SRAM.

   **[m:n]** To watch locations m through n inclusive. For example, sdram[0:3] watches locations 0 through 3 in SDRAM. And since a range can be specified in ascending or descending order, you could specify this watch as sdram[3:0].

   **[m:+n]** To watch location m plus the n locations following it. For example, sdram [5:+8] watches locations 5 through 8.

3. Optionally, you can specify a bit range to be watched. The format for a bit range is:

   **<m>** To watch only bit m of a state. For example, sdram[0]<12> watches only bit 12 of the sdram location 0.

   **<m:n>** To watch bits m through n of a state, with m being greater than n. For example, sdram[0:3]<12:10> watches bits 12 through 10 of sdram locations 0 through 3.

*Note:* SRAM and Scratchpad locations are 32-bit values, while SDRAM locations are 64-bit values.

## 2.11.13.2 Adding a Memory Watch

To add a memory watch:

1. Click **Add Watch** at the top of the **Memory Watch** window, or

   Right-click in the name or value column window and click **Add Watch** on the shortcut menu.

   The **Add Memory Watch** dialog box appears.

2. Select **Sram**, **Dram**, or **Scratchpad** under **Select the type of memory**.
3. Type the range of addresses to be watched in the **Select the range...** box.

   You can also specify a range of bits to be watched. By default, the entire location is watched.
4. Click **Add Watch**.

   When you are finished adding your watches,
5. Click **Close**.


### 2.11.13.3 Deleting a Memory Watch

To delete a memory watch:

1. Right-click the watch to be deleted in the **Memory Watch** window.
2. Click **Delete Watch** on the shortcut menu.

   or

1. Select the watch to be deleted.
2. Press DELETE.

### 2.11.13.4 Changing a Memory Watch

To change a memory watch:

1. Right-click the watch to be changed in the **Memory Watch** window and click **Edit Address** on the shortcut menu, or

   Double-click the mouse button on the watch to be changed.
2. Edit the address range (and bit range, if applicable).
3. Press ENTER.

### 2.11.13.5 Changing the Memory Watch Address Radix

To select whether memory addresses are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **Memory Watch** window.
2. Click **Hexadecimal Addresses** on the shortcut menu to display addresses in hexadecimal format or clear it to display in decimal format.

### 2.11.13.6 Changing the Memory Watch Value Radix

To select whether memory values are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **Memory Watch** window.
2. Click **Hexadecimal Data** on the shortcut menu to display values in hexadecimal format or clear it to display in decimal format.

### 2.11.13.7 Depositing Memory Data

#### Simulation Mode

In Simulation mode, you can change the value of any entry in the **Memory Watch** window.

#### Hardware Mode Restrictions

In Hardware mode, you can change only those entries that were added as a single memory location. You cannot change any value that is displayed as a result of adding a range of memory locations.

#### Values Changes

To change the value of a memory location that you are watching:

1. Right-click the value to be changed and click **Edit Value** on the shortcut menu, or

   Double-click the value to be changed.
2. Type the new value in either hexadecimal or decimal format.

   Hexadecimal values must be preceded by a ‘0x’.

## 2.11.14 About Execution Coverage

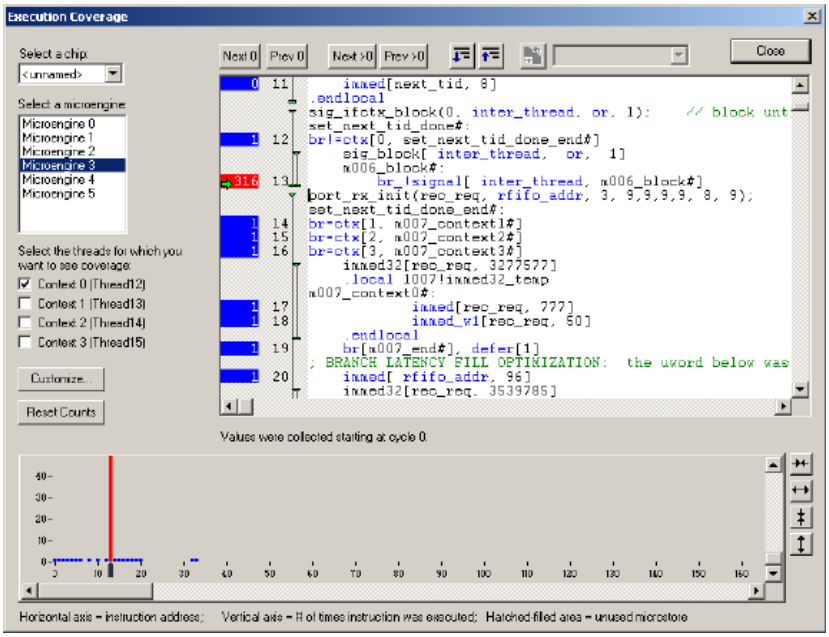
The code window in the **Execution Coverage** dialog box mimics the display in a thread window. That is, for a Microengine that contains C code, you can select a source view or a list view. In the source view you can select which source to display. In the source view, source lines show the sum of the number of times each generated instruction was executed. For example, if a source line generates three instructions and each instruction was executed 10 times, then the execution count displayed on that source line would be 30. Similarly, for assembled threads, the execution count for a collapsed macro reference is the sum of the execution counts for all instructions generated by the macro. The units for the horizontal axis on the bar graph remain instruction addresses.

To display the **Execution Coverage** dialog box (see Figure 2-7):

1. Stop  simulation (if necessary).
2. On the **Simulation** menu, click **Execution Coverage**.

   The **Execution Coverage** dialog box appears.

**Figure 2-7. The Execution Coverage WIndow**



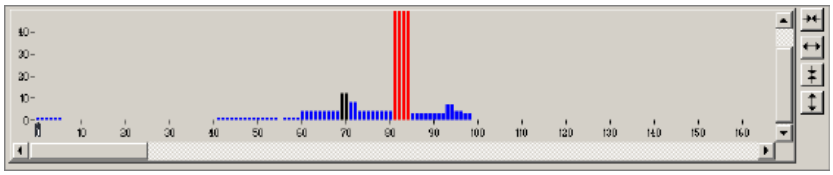

3. If your project contains multiple chips, select the chip whose coverage data you wish to view from **Select a chip** list in the upper left corner of the dialog box.
4. Select a Microengine from the **Select a Microengine** list.

The microcode that is loaded in that Microengine appears in the code window. This display is the same as is shown in the thread window.

### Execution Count

The number to the left of each instruction displays the number of times each instruction was executed. The background is color-coded to indicate a range of execution counts. (see Section 2.11.14.1).

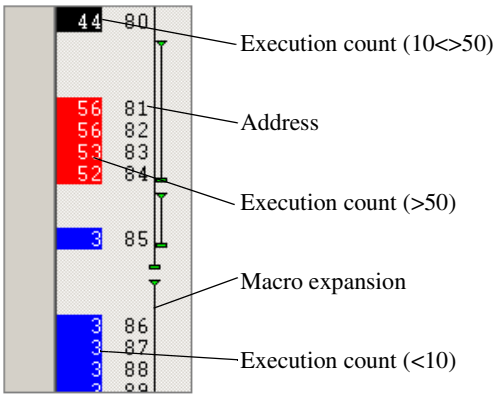
### Bar Graph



At the bottom of the dialog box is a bar graph that shows the execution coverage. The instruction addresses are represented along the horizontal axis, and the execution counts are represented by the vertical axis. The bars are color-coded using the same colors and ranges as in the code window.

By default, the execution counts are the total for all contexts in the Microengine. You can see the execution counts for any subset of contexts by selecting or clearing the four check boxes beneath the Microengine list.

The execution count for a collapsed macro reference is the sum of the execution counts of all instructions generated by the macro.




### 2.11.14.1 Changing Execution Count Ranges and Colors

By default, the colors and ranges for execution counts are:

| | |
|---|---|
| **Blue** | Instruction was executed less than 10 times. |
| **Black** | Instruction was executed between 10 and 50 times, inclusive. |
| **Red** | Instruction was executed more than 50 times. |

To change the colors and ranges, in the **Execution Coverage**, click **Customize**.

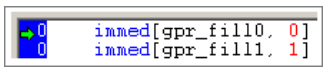
### 2.11.14.2 Displaying and Hiding Instruction Addresses

To toggle displaying and hiding instruction addresses in the code window:

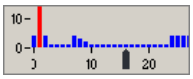
1. Right-click in the microcode window.
2. Select **Display Instruction Addresses** on the shortcut menu to display the addresses or clear to hide the addresses.

### 2.11.14.3 About Instruction Markers

To synchronize viewing between the code window and the bar graph, the Workbench displays an instruction marker. The 'current' instruction is marked in the code window by a horizontal green arrow in the leftmost gutter.




In the bar graph window, it is marked by a black vertical marker on the horizontal axis.




#### Marker Movement

Above the code window are four buttons that move the instruction marker:

| | |
|---|---|
| Next 0 | Moves the marker to the next instruction that has an execution count of 0. |
| Prev 0 | Moves the marker to the previous instruction that has an execution count of 0. |

| | |
|---|---|
| Next >0 | Moves the marker to the next instruction that has an execution count that is greater than 0. |
| Prev >0 | Moves the marker to the previous instruction that has an execution count that is greater than 0. |

You can also double-click an instruction in the code window or an address in the bar graph window and the marker moves to that instruction.

### 2.11.14.4 Miscellaneous Controls

Other controls in the **Execution Coverage** dialog box are:

| | | |
|---|---|---|
|  | **Expand macros**. | See Section 2.11.7.7. |
|  | **Collapse macros**. | See Section 2.11.7.7. |
|  | **Toggle views**. | See Section 2.11.7.4. |
|  | **Select the source file to view**. | |

### 2.11.14.5 Scaling the Bar Graph

To the right of the bar graph window are four buttons for scaling. They are:

| | |
|---|---|
|  | **Horizontal zoom out**. |
|  | **Horizontal zoom in**. |
|  | **Vertical zoom out**. |
|  | **Vertical zoom in**. |

### 2.11.14.6 Resetting Execution Counts

By default, execution counting starts at the first simulation cycle. If you have initialization code that you don't want included in the counts:

1. Run the simulation until it completes the initialization.
2. On the **Simulation** menu, click **Execution Coverage**.

   The **Execution Coverage** dialog box appears.
3. Click **Reset Counts**.

Execution counting restarts from the cycle at which you clicked Reset.

## 2.11.15 About Performance Statistics

The Workbench provides the ability to gather and display statistics on simulation performance. Statistics gathering is available only in Simulation mode.

### 2.11.15.1 Displaying Statistics

To display a **Statistics Summary**:

1. Stop debugging (if necessary).
2. On the **Simulation** menu, click **Statistics**.

   The **Statistics** dialog box appears.
3. Click the **Summary** tab.

The **Summary** page shows the percentage of time that each Microengine and memory unit is active and the rate that this activity represents.

4. Click the **Microengine** tab.

The Microengine statistics page contains a multicolumn hierarchical tree displaying the statistics. The first column identifies the component for which the statistics apply. The next four columns show the percentage of time that the component was executing, aborted, stalled, idle, and swapped out. You can expand and collapse the tree by clicking on the + sign to the right of a component, or by double-clicking on the component.

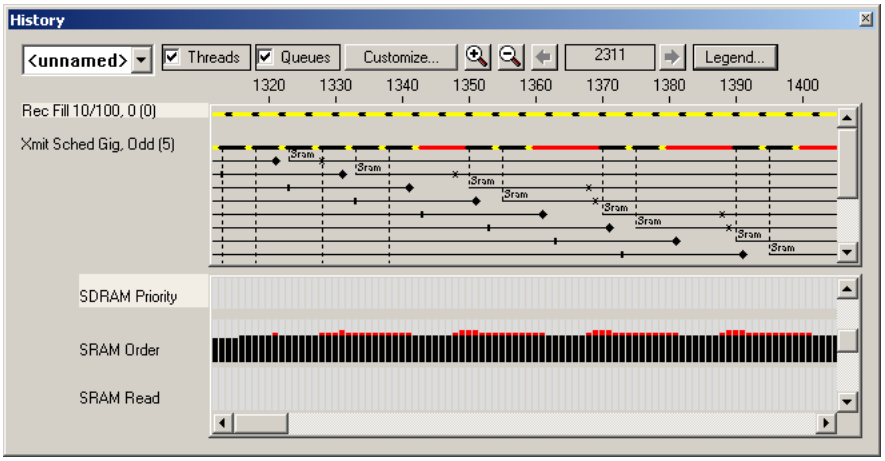
### 2.11.15.2 Resetting Statistics

By default, statistics are gathered starting at cycle 1 of a simulation. However, you can reset the statistics at any time. The statistics are then gathered from the current cycle forward.

To reset statistics, on the **Simulation** menu, click **Reset Statistics**.

## 2.11.16 About Thread and Queue History

Thread and queue history is a powerful feature that enables you to look at the status of all 24 threads and all SDRAM and SRAM queues in a chip at the same time. It provides a high level view of how your microcode is executing, enabling you to quickly locate performance bottlenecks.




Threads and queues are displayed on a timeline that represents the number of cycles executed. A thread's history is depicted by line segments that change color depending on whether an instruction is executing or aborted, if the thread is stalled, or the Microengine is idle. A queue history is depicted by vertical bars whose height indicates how many entries are in the queue at a given cycle. You select which chip's history is displayed using the box in the upper left corner of the **History** window. Chip selection is synchronized with chip selection in the **Thread Status**, **Queue Status** and **Memory Watch** windows.

### 2.11.16.1 Displaying the History Window

- On the **View** menu, click **Debug Windows**, then select **History**, or
  Click the  button on the **View** toolbar.

This toggles visibility of the **History** window. You must be in debug mode to view history.

#### Hardware Debugging Restrictions

When debugging in the hardware configuration, thread and queue history is not supported.

### 2.11.16.2 Scaling the Display

To control the horizontal scale for the history display, use the zoom in  and zoom out  buttons.

### 2.11.16.3 Displaying Thread History Lines

You can control whether or not a specific thread's history line is displayed. This allows you to limit the display to only those threads that you are interested in. By default, all history lines are displayed.

To hide a thread's history line:

1. Right-click the thread's name or its history line.
2. Click **Hide** 'threadname' on the shortcut menu, where 'threadname' is the name of the thread you clicked on.

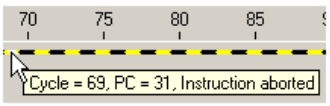
   You can also right-click the thread's name in **ThreadView** and click **Hide Thread History** on the shortcut menu.

To redisplay a hidden thread's history:

1. Right-click in the thread history subwindow.
2. Click **Display Thread** then click the thread's name on the shortcut menu.

   You can also right-click the thread's name in **ThreadView** and click **Display Thread History** on the shortcut menu.

To get details about a thread's history, position the cursor over the history line and wait for a moment. The Workbench displays the cycle count, PC, and the instruction state in a pop-up window beneath the cursor.




### 2.11.16.4 Displaying Code Labels

The thread history window supports the viewing of microcode labels along a thread's history line. This helps in determining what code is being executed during a certain cycle.

To specify which code labels to displayed, do the following:

1. In the **History** window, click **Customize**.

   The **Customize Thread History** dialog box appears.

2. Click the **Code Labels** tab.
3. In the **Select Microengine** box is a list of all the Microengines in the project. Click a Microengine to display all the labels in the microcode associated with that Microengine.

   :

4. In the **All labels in MicroEngine's microcode** box, select the labels to be displayed in the **History** window by clicking the label, and then clicking **Add**.

   The Labels to be displayed in thread history box lists all the code labels you have selected to be displayed on the selected Microengine's history lines.

5. Continue this procedure for each Microengine for which you want code labels displayed.
6. To delete a label from the display list, select the label in the rightmost box and then click the button.
7. Click **OK** to close the **Customize Thread History** dialog box.

### Code Labels for a Thread

Displaying code labels on a particular thread's history line is controlled via shortcut menus in the **History** window.

- To display code labels for a thread, right-click on the thread name or on its history line and check **Display Code Labels for** 'threadname' on the shortcut menu, where 'threadname' is the name of the thread you clicked on.
- To hide code labels for a thread, right-click and uncheck Display Code Labels for 'threadname'.

### Code Labels for a Microengine

Whether or not code labels are displayed on a particular Microengine's history line is controlled by shortcut menus in the **History** window.

To display or hide code labels for a Microengine:

1. Right-click the thread name or on its history line.
2. Click **Display Code Labels for**.
3. Click the Microengine you wish to display or hide code labels.

### Code Labels

You separately control whether or not code labels are displayed at all.

- Right-click and select **Enable Code Label Display** on the shortcut menu.

  The Workbench displays code labels for those threads that you have selected as having code labels displayed.

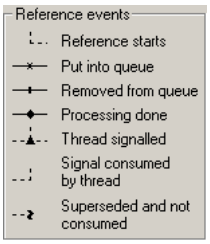
- If you right-click and clear **Enable Code Label Display** on the shortcut menu, the Workbench hides all code labels.

### 2.11.16.5 About Reference History

The **Thread History** window supports the viewing of reference history lines underneath the history lines of the thread issuing the reference.

The reference line starts at the cycle when the reference is issued and ends at the cycle when the reference's signal is consumed by the thread. If the thread doesn't request a signal, then the reference line ends at the cycle when processing is complete.

On a reference line, the Workbench displays markers at the cycles when reference events occur.



Reference Events

**X** Displayed at the cycle when the reference is put into the queue of the unit being referenced.

**Tick mark** Displayed at the cycle when the unit removes the reference from the queue.

**Diamond** Displayed at the cycle when the unit finishes processing the reference.

**Up-arrow** Displayed at the cycle when the unit signals the thread that the reference is completed.

**Dotted line** Shown from the reference line to the thread's history line at the cycle when the thread consumes the signal.

**Thick vertical squiggle**
Displayed when a reference's signal has been superseded by another reference's signal before the signal was consumed. When this occurs, if the signalled thread eventually consumes the signal, it will be consuming the superseding reference's signal, not the original reference's signal.

Interthread references are displayed in two sections:

- The reference creation section is displayed underneath the signalling thread, with the name of the signalled thread shown above the reference line.
- The reference consumption section is displayed underneath the signalled thread, with the name of the signalling thread shown above the reference line.

To get details about a reference:

- Position the cursor over the reference line and wait for a moment.

  The Workbench displays the reference command, the address it is accessing, and the number of longwords being referenced in a pop-up window beneath the cursor.

For details on how to customize the reference line colors, see Section 2.11.16.7.

Only SDRAM, SRAM, interthread and IX Bus Interface references are available.

### 2.11.16.6 Displaying References

Whether or not references are displayed on a particular thread's history line is controlled via shortcut menus in the **History** window.

To display references for a thread:

1. Right-click the thread name or its history line.

2. Click **Display References for** ‘threadname’ on the shortcut menu, where ‘threadname’ is the name of the thread you clicked on. To hide references for a thread, right-click and clear **Display References for** ‘threadname’.

To display or hide references for all threads in a Microengine:

- Right-click in the thread history area and click **Display References for**, then click **MicroEngine *n*** on the shortcut menu, where n is the number of the Microengine for which you want all thread references displayed.
- To hide references for all threads in a Microengine, right-click in the thread history area and click **Hide References for**, then click **Microengine *n***.

You separately control whether or not references are displayed at all.

- If you right-click and click **Enable Reference Display** on the shortcut menu, the Workbench displays references for those threads that you have selected as having references displayed.
- If you right-click and clear **Enable Reference Display** on the shortcut menu, the Workbench hides all references.

### 2.11.16.7 Changing Thread History Colors

By default, the Workbench displays a thread history line in:

 if an instruction is executing

 if it is aborted

 if the thread is stalled

 if the Microengine is idle

Reference lines are displayed in black.

To change the colors for the thread history and reference lines, do the following:

1. In the thread history window, click **Customize**, or

   On the **Simulation** menu, click **Simulation Options**.
2. Click the **Colors** tab.
3. Click the color button next to the item whose color you want to change.
4. Select the new color.
5. Click **OK** after specifying the colors you want.

*Note:* For consistency, the execution state colors also apply to the instruction markers that are displayed in the thread windows.

### 2.11.16.8 Displaying the History Legend

To see a legend of the thread history colors and the reference event markers, click the **Legend** button in the **History** window. The modeless History Legend dialog box appears.

### 2.11.16.9 Tracing Instruction Execution

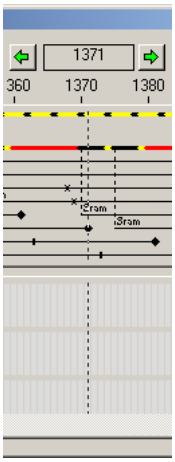
To view the instruction that a thread was executing at a given cycle count,

1. Right-click the thread's history line at the cycle count in which you are interested.
2. Click **Go To Instruction** on the shortcut menu.

This opens or activates the thread's code window and scrolls it to show the requested instruction.

- If the requested instruction is displayed, the history instruction marker appears on the appropriate instruction line.
- If the requested instruction is not displayed because it is in a collapsed macro reference, the history instruction marker is displayed on the line with the macro reference.
- If the thread is compiled and the source view is being displayed, the history instruction marker is displayed on the line that generates the instruction.
- If the thread is compiled and the list view is being displayed, the history instruction marker is displayed on the appropriate instruction line.

The thread history window contains a cycle marker that marks a particular cycle count using a vertical dashed line cutting across all displayed history lines (see image at left). The cycle count at the cycle marker's position is reported in the box located between the right and left green arrow buttons in the **History** window.




There are several ways to move the cycle marker:

- To immediately move the cycle marker to a given cycle count, double-click the **History** window at the cycle where you want the marker.
- To drag the cycle marker, press the left mouse button on the marker, drag to the desired cycle and release the mouse button. As you drag, the marker snaps to cycle count positions and the cycle count is displayed.
- To move the cycle marker to the next cycle, click the button labeled with the green right arrow.
- To move the cycle marker to the previous cycle, click the button labeled with the green left arrow.

If a thread's code window is opened, movement of the cycle marker scrolls the window to show the instruction being executed by the thread at that cycle count. The instruction is marked with a green arrow in the window's gutter. This enables you to trace past program flow by going to a specific cycle and incrementing the cycle marker.

### 2.11.16.10 About Queue History

You can control whether or not a specific queue's history is displayed. This allows you to limit the display to only those queues that you are interested in. By default, all queues are displayed.

#### Hide

To hide a queue's history:

1. Right-click the queue's name or its history.

2. Click **Hide** ‘queuename’ on the shortcut menu, where ‘queuename’ is the name of the queue you clicked on.

#### Display

To display or redisplay a hidden queue’s history:

1. Right-click in the queue history subwindow
2. Click **Display Queue** then click the queue’s name on the shortcut menu.

#### Queue’s Contents

To get information about a queue’s contents:

1. Position the cursor over a vertical bar and wait for a second.
2. The Workbench displays the number of entries in the queue and the size of the queue in a pop-up window beneath the cursor.

#### Queue’s Contents in Detail

To get detailed information about the contents of the queues:

- Right-click and click **Show Queue Status** on the shortcut menu.

  The Queue Status window appears showing details of all the queues. (See Section 2.11.17 for more information on the Queue Status window.)

### 2.11.16.11 About History Collecting

You can control whether or not history data is collected. Not collecting history data improves simulation performance by approximately 10 percent.

#### Disable

To disable history collecting,

1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.
2. Click the **History** tab.
3. Clear **Enable history collecting**.

#### Enable

To collect history:

1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.
2. Click the **History** tab.
3. Select **Enable history collecting**.
4. Select:
   a. **Collect Thread History**, or
   b. **Collect Queue History**, or

c. Both, depending on what history you want collected.

5. Specify how many cycles of history you want collected by typing a number in the corresponding box.
6. If you want to have history collected throughout the simulation, click **Always collect**.
7. If you want to run the simulation for a time before starting to collect history, click **Start collecting at cycle** and type the cycle count at which you want collecting to start.
8. If you want the simulation to stop after the history has been collected, select **Stop simulation after collecting the specified number of cycles**.

*Note:* These options must be specified before you start debugging. If you are already in debug mode, these selections are disabled.

## 2.11.17 About Queue Status

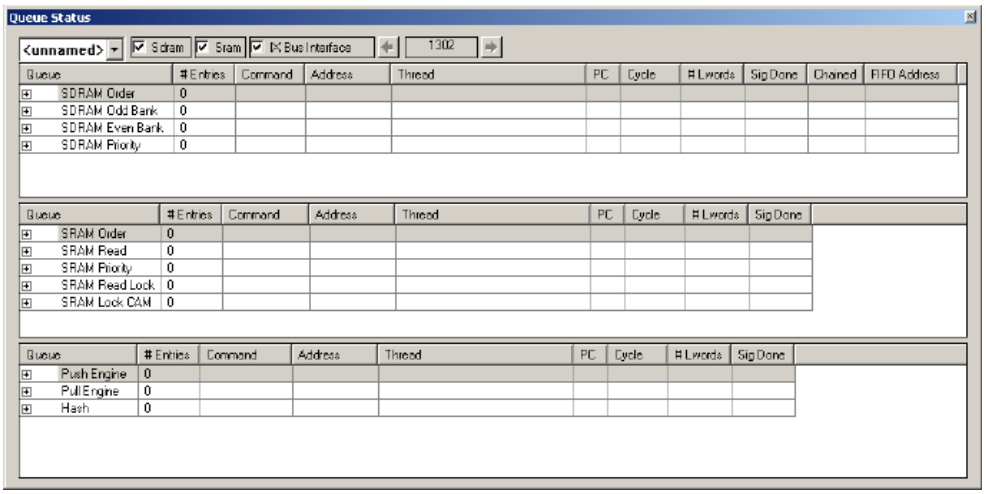
The queue status window provides current and historical information on the contents of the SDRAM, SRAM and IX Bus Interface queues, with one subwindow for each queue type. Each queue type has an associated check box at the top of the queue status window to control whether or not it's subwindow is visible.

- To display the Queue Status window, on the **View** menu, click **Debug Windows**, then click **Queue Status**, or

  Click the  button on the **View** toolbar.

  The Queue Status window appears (see Figure 2-8).

**Figure 2-8. The Queue Status Window**




Select which chip's queue status is displayed using the list in the upper left corner. Chip selection is synchronized with chip selection in the **Memory Watch, Thread Status** and **History** windows.

The number of entries in the queue is displayed for each queue in the SDRAM, SRAM and IX Bus Interface.

To examine the references that are in a queue, expand the queue's tree item by clicking on the + symbol or double-clicking on the item. For each reference in the queue, the Workbench displays:

- The type of reference.
- The address being referenced.
- Which thread made the reference.
- The PC of the instruction which made the reference.
- The cycle count at which the reference was made.
- The number of longwords being referenced.
- Whether a signal will be generated when the reference is completed.

In addition, the SDRAM references show:

- Whether the reference is chained
- The destination FIFO address (if applicable).

#### Instruction Cross-reference

If you right-click a reference and click **Go To Instruction** on the shortcut menu, click the Workbench opens the appropriate thread window and displays the microcode instruction that issued the reference. A purple marker in the left margin of the thread window marks the instruction. The reference is also highlighted with the same marker in the queue status window.

### 2.11.17.1 About Queue Status History

When the simulation stops, the queue status window is automatically updated to show the current queue contents. If you enabled collecting of queue history, you can also review the contents of the queues for previously executed cycles.

To do this:

- Click the right and left arrows at the top of the queue status window.

  The number between the arrows shows the cycle count at which the queue contents occurred. This historical cycle count is synchronized with the corresponding cycle count in the **History** window. Changing either one also changes the other. The allows you to move the graphical cycle marker in the **History** window to a specific cycle and view the queue contents in relation to the thread history.

### 2.11.17.2 Setting Queue Breakpoints

You can set a breakpoint on a queue to have simulation stop when the queue rises to or falls below a specified threshold.

To do this:

1. Right-click the queue name and click **Insert/Remove Breakpoint** on the shortcut menu.

   The **Queue Breakpoint** dialog box appears.

2. By default, the breakpoint is enabled and triggers when the queue rises to the default threshold for the queue. You can change the trigger threshold by changing the number in the Threshold box.

3. By selecting or clearing the check boxes, you can change the breakpoint properties.

When a breakpoint is set and enabled on a queue, a solid red breakpoint symbol is displayed to the right of the queue name.

To disable a breakpoint:

1. Right-click the queue name.
2. Click **Enable/Disable Breakpoint** on the shortcut menu.

   A disabled breakpoint is indicated by an unfilled breakpoint symbol.

To enable a breakpoint:

1. Right-click the queue name.
2. Click **Enable/Disable Breakpoint** or **Insert/Remove Breakpoint** on the shortcut menu.

To remove an enabled breakpoint:

1. Right-click the queue name.
2. Click **Insert/Remove Breakpoint** on the shortcut menu.

To change a breakpoint's properties:

1. Right-click the queue name.
2. Click **Breakpoint Properties** on the shortcut menu.

   The **Queue Breakpoint** dialog box reappears where you can change properties or click **Remove** to remove the breakpoint.

## 2.11.18 About Thread Status

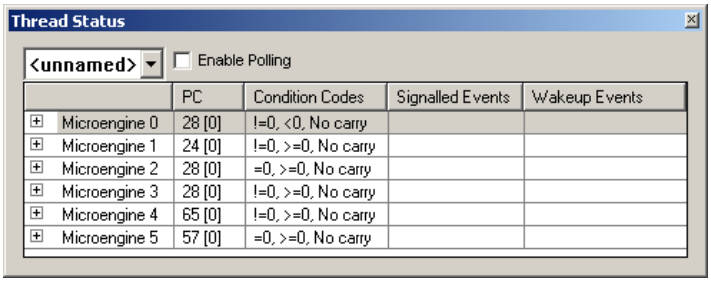
The **Thread Status** window provides static or snapshot information on the status of each thread in a selected chip in your project. For each thread, the following information is displayed:

- Current instruction address.
- The list of events for which the thread is waiting.
- The list of events that have been signaled to the thread.

- To display the **Thread Status** window, on the **View** menu, click **Debug Windows**, then click **Thread Status** (see Figure 2-9), or

  Click the  button the **View** toolbar.

**Figure 2-9. The Thread Status Window**




In each Microengine, an arrow appears to the left of the thread that is currently executing or that is scheduled to resume execution when the Microengine resumes execution.

Select which chip's status is displayed using the box in the upper left corner of the **Thread Status** window. Chip selection is synchronized with chip selection in the memory watch, queue status and history windows.

### View

You can control which threads are displayed by expanding and collapsing the Microengine entries in the status tree. You can expand the tree so that all threads of the selected chip are displayed by right-clicking and selecting **Expand All** on the shortcut menu.

### Update

The status display is updated whenever Microengine execution stops, for example, when you stop execution or when you hit a breakpoint.

### Polling

You can also have the Workbench poll the threads and update the status at regular intervals. To enable or disable thread status polling and to change the polling interval:

1. On the **Debug** menu, click **Status Polling**, or

   Right-click within the **Thread Status** window and click **Status Polling** on the shortcut menu.

   The **Status Polling** dialog box appears.

2. Select **Poll thread status** to enable polling or clear it to disable polling.

*Note:* You can also enable and disable polling in the Thread Status dialog box by selecting or clearing Enable Polling.

### Polling Interval

If you enable polling, specify the polling interval by:

- Typing the number of seconds between polls in the Polling interval (sec) box. You can also use the spin controls to increment or decrement the number in the box.

  The value that you type in must be an integer.

## 2.11.19 About IX Bus Device Simulation

The Workbench provides simulation of IX Bus devices as well as simulation of network traffic. To simulate devices and network traffic you need to:

1. Configure the devices on the IX Bus (or busses if you have multiple IXP12nn chips).

   This involves specifying how many devices are on the bus as well as the characteristics of each device.

2. Create one or more data streams (see section Section 2.11.20).

   These streams can consist of Ethernet frames or ATM cells.

3. Assign one or more data streams or a network traffic DLL to each device port that you want to receive network traffic.

4. Specify the options under which you want the traffic simulation to operate.

While you are running your simulation, you can observe the status of the IX Bus and its devices in the **IX Bus Device Status** window (see Section 2.11.23.4).

### 2.11.19.1 Configuring IX Bus Devices

To configure the devices on the IX Bus:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Devices and Ports**.

   The **IX Bus Devices and Ports** dialog box appears.

2. If there is more than one IX Bus in your system configuration, select the IX Bus you wish to configure in the **Select an IX bus** box.

#### Adding a Device

To add a device to the selected IX Bus:

1. Click **Add Device**.

   The **Add IX Bus Device** dialog box appears.

2. Type the number of ports that the device supports and optionally enter a brief device description, for example, OctalMAC.

#### Setting Default Port Characteristics

You can also customize the default port characteristics. These characteristics, in the **Default port characteristics** area, are:

| | |
|---|---|
| **Data rate (Mbits/sec)** | Specifies the rate at which data is taken from the network and inserted into the port's receive (Rx) buffer and the rate at which data is taken from the port's transmit (Tx) buffer and put onto the network. |
| **Interpacket gap (nsec)** | Specifies the amount of time between packets when receiving packets from and transmitting packets to the network. |
| **Number of pruned bytes** | Specifies the number of bytes of packet data that is normally processed by the device and not passed to the IXP12nn, such as an Ethernet preamble. |

| | |
|---|---|
| **Receive buffer size** | Specifies the number of bytes in the receive buffer. The receive buffer holds the data received from the network until the IXP12nn reads it from the port. |
| **Receive ready threshold** | Specifies number of bytes that must be in the port's receive buffer in order for the port to assert its receive ready (RxRdy) bit. This signals the IXP12nn that it can select the port and request data from it. |
| **Strip trailer on receive** | Specifies that you want the trailer stripped from the end of each frame before it is received by the IXP12nn. |
| **Fast port** | By selecting this check box, you specify that the port is to assert the FastRx1 or FastRx2 signal when it asserts RxRdy. **Note:** You are limited to two fast ports in your simulation. One has to be Fast port 1 and the other has to be Fast port 2. Both fast ports can be on one device or each can be on separate devices. |
| **Transmit buffer size** | Specifies the number of bytes in the transmit buffer. The transmit buffer holds the data transmitted by the IXP12nn until it is transmitted onto the network. |
| **Transmit ready threshold** | Specifies number of free bytes that must be available in the port's transmit buffer in order for the port to assert its transmit ready (TxRdy) bit. This signals the IXP12nn that it can select the port and transmit data to it. |
| **Transmit send threshold** | Specifies the number of bytes that must be in the transmit buffer before the data in the buffer is transmitted onto the network. Data is transmitted even if the number of bytes is below the threshold provided there is at least one end-of-packet (EOP) in the buffer. |
| **Maximum packets allowed in buffer** | Specifies the maximum number of packets that can be in the transmit buffer. When this number of packets is in the buffer, the port will not assert TxRdy even if the number of free bytes exceeds the transmit ready threshold. |
| **Number of bytes to strip** | Specifies the number of bytes to be stripped from the end of each received frame. |
| **Append trailer on transmit** | Specifies that you want an trailer appended to the end of each frame that is transmitted by the IXP12nn. |
| **Number of bytes to append** | Specifies the number of bytes to be appended to the end of each transmitted frame. |

### Standard Templates

As an alternative to manually entering parameter values, you can select a template from the template list. When you select a template, all parameters are updated with the values that are stored with that template. The Workbench comes with two pre-defined templates for the IXF1002 and the IXF440 MAC devices.

To load a standard template:

1. On the **Simulation** menu, click **IX Bus Simulator Devices,** and then click **Devices and Ports**.

The **IX Bus Devices and Ports** dialog box appears.

2. Click **Add Device**.

   The **Add IX Bus Device** dialog box appears.

3. From the list in the upper-left corner, select either **Standard IXF1002**, or **Standard IXF440**.

### Template Customization and Save

In the **Add IX Bus Device** dialog box, you can create your own template by:

1. Setting the parameters you wish to save.
2. Clicking **Save Template**.

   The **Save Template** dialog box appears.

3. Type the name for your template.
4. Specify a file path where the template will be saved.

   You can browse to a file or folder by clicking the ... button.

5. Click **OK** when done.
6. In the **Add IX Bus Device** dialog box, the template that you just created appears in the list in the upper-left corner.

### Templates Import

The default file extension for template files is ‘wbt’. If you have template files that you copied from other sources, you import those into your Workbench configuration.

To import a template(s) from the **Add IX Bus Device** dialog box:

1. Click **Import Template**.

   The **Import IX Bus Device Template** dialog box appears.

2. Browse to the file(s) you want to import.
3. Select them then click **OK**.

### Device Removal

To remove a device from the bus:

1. On the **Simulation** menu, click **IX Bus Simulator Devices,** and then click **Devices and Ports**.

   The **IX Bus Devices and Ports** dialog box appears.

2. Select the device you want to remove.
3. Click **Remove Device**.

Since device numbering always start at 0, removing a device may cause renumbering of the remaining devices.

### Port Characteristics Edit

To edit an individual port’s characteristics:

1. In the **IX Bus Devices and Ports** dialog box, select the port you want to edit.
2. Click **Edit Port**.

The **Configure Port** dialog box appears.

3. Modify any of the **Port characteristics.**
4. When you have finished configuring all devices and ports, click **OK**.

Any changes that you have made now appear in the corresponding column of the port.

## 2.11.20 About Data Streams

Data streams are used to simulate network traffic.

### 2.11.20.1 Creating and Editing a Data Stream

You can create and edit the following date streams:

| | |
|---|---|
| **ATM** | See Section 2.11.21.1. |
| **Custom Ethernet IP** | See Section 2.11.21.2. |
| **Ethernet IP** | See Section 2.11.21.3. |
| **Ethernet TCP/IP** | See Section 2.11.21.4. |
| **PPP TCP/IP** | See Section 2.11.21.5. |

### 2.11.20.2 Deleting a Data Stream

To delete a data stream:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams**.

   The **Data Streams** dialog box appears.
2. Click the data stream that you want to delete.
3. Click **Delete Stream**.

*Note:* The Workbench only deletes the data stream from the project, not from the folder. If you delete in error, click Import Stream(s) to retrieve it.

### 2.11.20.3 Importing a Data Stream

To import a data stream from a previously saved file:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams**.

   The **Data Streams** dialog box appears.
2. Click **Import Stream**(s).

   The **Import Stream** dialog box appears.
3. Browse to the desired folder and select one or more stream files (.strm).
4. Click **OK** to import the selected files.

### 2.11.20.4 Copying a Data Stream

Copying a data stream then editing the copy gives you a quick way to create a new data stream that is similar to an existing data stream.

To copy an existing data stream:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams**.

   The **Data Streams** dialog box appears.
2. Click the data stream that you want to copy.
3. Click **Copy Stream**.

   The **Stream Name** dialog box appears.
4. Type the name of the new data stream.
5. Click **OK**.

   The **Specify file...** dialog box appears.
6. Select the folder where you want to save the stream.
7. Click **OK**.

   The stream appears (or reappears if you did not change the name) in the Data Streams dialog box. It has all the characteristics of the stream copied.

### 2.11.20.5 Assigning an IX Bus Device Port

To assign data streams to IX Bus device ports:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, and then click **Data Streams**.

   The **Data Streams** dialog box appears.
2. Click the data stream that you want to assign.
3. Click **Assign Port I/O**.

See Section 2.11.21.14 for more detailed information.

## 2.11.21 Creating and Editing Different Data Streams Types

This section explains how to create five different data streams. They are:

| | |
|---|---|
| **ATM** | See Section 2.11.21.1. |
| **Custom Ethernet IP** | See Section 2.11.21.2. |
| **Ethernet IP** | See Section 2.11.21.3. |
| **Ethernet TCP/IP** | See Section 2.11.21.4. |
| **PPP TCP/IP** | See Section 2.11.21.5. |

### 2.11.21.1 Creating and Editing an ATM Data Stream

An ATM Protocol Data Unit (PDU) comprises the unsegmented components of ATM data:

- The ATM Header
- An AAL5 trailer
- An optional LLC/SNAP header
- An IP packet payload

#### Creating

To create an ATM data stream:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams**.

   The **Data Streams** dialog box appears.

2. Click **Create Stream**.

   The **Create Steam** dialog box appears.

3. Type the name of the ATM stream in the **Stream name** box.
4. Select **ATM** from the **Stream type** list (if not already selected).
5. Click **Continue**.

   The **ATM Stream** dialog box appears.

To create a PDU:

1. Click **Create PDU(s)**.

   The **Create AAL5 PDU** dialog box appears.

2. Type the values you want for the **ATM header**.

   This header is prepended to each segmented ATM cell. If you select **Automatic** for **PTI**, then the box will contain zero for all cells except for the last one, where the box will contain a one.

3. For **RFC1483 options**, you can select **LLC/SNAP**, which prepends a header to the packet data, or **VCMUX**, which does not prepend a header. The optional header plus the packet data constitute a CS-DSU information box. Currently, an AAL5 trailer is always appended to the CS-DSU information box.
4. If you want to encapsulate a single IP packet, click the **Single Packet** option.
   - — Click **IP Header** to specify the fields of the IP header (see Section 2.11.21.9),
   - — Click **IP Payload** to specify the data payload for the IP packet (see Section 2.11.21.12).
5. If you want to encapsulate a pool of packets, click the **Multiple packets from pool** option. Select a packet pool from the list of available pools.
6. To import a previously created packet pool, click **Import Pool**.
7. To create a new pool, click **Create Pool**.

   The **IP Packet Pool** dialog box appears. Go to Section 2.11.21.6 to create the IP packet pool.

8. Click **Create Packet(s)** to create a PDU(s) for each packet in the selected pool.
9. Click **IP Header** to specify IP Header information (see Section 2.11.21.9).

The created PDUs are added to the ATM data stream. The dialog box remains active so you can change settings and create additional PDUs.

10. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.21.13).
11. When you are finished creating PDUs, click **Close**.
12. When you are finished creating the data stream, click **OK**.

#### Editing

To edit an ATM stream:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams.**

   The **Data Streams** dialog box appears.
2. Select an ATM stream that you want to edit.
3. Click **Edit Stream**.

   The **ATM Stream** dialog box appears.
4. Here you can:
   - **Edit PDUs** (similar to creating—see previous section).
   - Delete a PDU by selecting the PDU and clicking **Delete PDU**.
   - Change the order of the PDUs using the  or  arrows.
5. Click **OK** when done.

### 2.11.21.2 Creating and Editing a Custom Ethernet IP Data Stream

#### Creating

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams**.

   The **Data Streams** dialog box appears.
2. Click **Create Stream**.

   The **Create Stream** dialog box appears.
3. Type the name of the stream in the **Stream name** box.
4. Select **Custom Ethernet IP** from the **Stream type** list.
5. Click **Continue**.

   The **Custom Header Size** dialog box appears.
6. Type the number of bytes to be in the custom header.
7. Click **OK**.

   The **Custom Ethernet IP Data Stream** dialog box appears.
8. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.
2. Click **Custom Header** (see Section 2.11.21.7).

3. Click **Ethernet Header** (see Section 2.11.21.8).
4. Click **IP Header** (see Section 2.11.21.9).
5. Click **Data Payload** (see Section 2.11.21.12).
6. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.21.13).
7. Type the number of frames you want to create in the **Number of new...** box.
8. Click **Create**.

   The number of frames you specified is created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.
9. When you are finished creating frames click **Close**.
10. When you are finished creating the data stream, click **OK**.

    The **Save** dialog box appears.
11. Type in the file name if you want to change it.
12. Browse to the folder where you want to save the file.
13. Click **Save**.
14. In the **Data Streams** dialog box, click **OK.**

#### Editing

To edit a Custom Ethernet IP data stream:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams.**

   The **Data Streams** dialog box appears.
2. Select stream that you want to edit.
3. Click **Edit Stream**.

   The **Custom Ethernet IP Data Stream** dialog box appears.
4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:

   | | |
   |---|---|
   | **Custom Header** | See **Section 2.11.21.7**. |
   | **Ethernet Header** | See Section 2.11.21.8. |
   | **IP Header** | See Section 2.11.21.9. |
   | **Data Payload** | See Section 2.11.21.12. |
6. Click **Delete Frame** to delete the selected frame.
7. Click the  and  arrows to change the order of the frames.
8. Click **OK** when done.

### 2.11.21.3 Creating and Editing an Ethernet IP Data Stream

#### Creating

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams**.

   The **Data Streams** dialog box appears.

2. Click **Create Stream**.

   The **Create Stream** dialog box appears.

3. Type the name of the stream in the **Stream name** box.
4. Select **Ethernet IP** from the **Stream type** list.
5. Click **Continue**.

   The **Ethernet IP Data Stream** dialog box appears.

6. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.
2. Click **Ethernet Header** (see Section 2.11.21.8).
3. Click **IP Header** (see Section 2.11.21.9).
4. Click **Data Payload** (see Section 2.11.21.12).
5. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.21.13).
6. Type the number of frames you want to create in the **Number of new...** box.
7. Click **Create**.

   The number of frames you specified is created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.

8. When you are finished creating frames click **Close**.
9. When you are finished creating the data stream, click **OK**.

   The **Save** dialog box appears.

10. Type in the file name if you want to change it.
11. Browse to the folder where you want to save the file.
12. Click **Save**.
13. In the **Data Streams** dialog box, click **OK.**

### Editing

To edit an Ethernet IP data stream:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams.**

   The **Data Streams** dialog box appears.

2. Select stream that you want to edit.
3. Click **Edit Stream**.

   The **Ethernet IP Data Stream Dialog** box appears.

4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:

| | |
|---|---|
| **Ethernet Header** | See Section 2.11.21.8. |
| **IP Header** | See Section 2.11.21.9. |
| **Data Payload** | See Section 2.11.21.12. |

6. Click **Delete Frame** to delete the selected frame.
7. Click the  and  arrows to change the order of the frames.
8. Click **OK** when done.

*Note:* The Workbench puts a four byte CRC value at the end of each packet that is included in the byte count.

## 2.11.21.4 Creating and Editing an Ethernet TCP/IP Data Stream

### Creating

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams**.

   The **Data Streams** dialog box appears.
2. Click **Create Stream**.

   The **Create Stream** dialog box appears.
3. Type the name of the stream in the **Stream name** box.
4. Select **Ethernet TCP/IP** from the **Stream type** list.
5. Click **Continue**.

   The **Ethernet TCP/IP Data Stream** dialog box appears.
6. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.
2. Click **Ethernet Header** (see Section 2.11.21.8).
3. Click **IP Header** (see Section 2.11.21.9).
4. Click **TCP Header** (see Section 2.11.21.10).
5. Click **Data Payload** (see Section 2.11.21.12).
6. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.21.13).
7. Type the number of frames you want to create in the **Number of new...** box.
8. Click **Create**.

   The number of frames you specified is created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.
9. When you are finished creating frames click **Close**.
10. When you are finished creating the data stream, click **OK**.

    The **Save** dialog box appears.
11. Type in the file name if you want to change it.
12. Browse to the folder where you want to save the file.
13. Click **Save**.
14. In the **Data Streams** dialog box, click **OK**.

### Editing

To edit an Ethernet TCP/IP data stream:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams.**

   The **Data Streams** dialog box appears.
2. Select stream that you want to edit.
3. Click **Edit Stream**.

   The **Ethernet TCP/IP data Stream Dialog** box appears.
4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:

   | | |
   |---|---|
   | **Ethernet Header** | See Section 2.11.21.8. |
   | **TCP Header** | See Section 2.11.21.10. |
   | **IP Header** | See Section 2.11.21.9. |
   | **Data Payload** | See Section 2.11.21.12. |
6. Click **Delete Frame** to delete the selected frame.
7. Click the  and  arrows to change the order of the frames.
8. Click **OK** when done.

## 2.11.21.5 Creating and Editing a PPP TCP/IP Data Stream

### Creating

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams**.

   The **Data Streams** dialog box appears.
2. Click **Create Stream**.

   The **Create Stream** dialog box appears.
3. Type the name of the stream in the **Stream name** box.
4. Select **PPP TCP/IP** from the **Stream type** list.
5. Click **Continue**.

   The **PPP TCP/IP Data Stream** dialog box appears.
6. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.
2. Click **PPP Header**. Type the **Protocol** value and click 8-bit or 16-bit.
3. Click **IP Header** (see Section 2.11.21.9).
4. Click **TCP Header** (see Section 2.11.21.10).
5. Click **Data Payload** (see Section 2.11.21.12).
6. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.21.13).
7. Type the number of frames you want to create in the **Number of new...** box.

8. Click **Create**.

   The number of frames you specified is created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.

9. When you are finished creating frames click **Close**.
10. When you are finished creating the data stream, click **OK**.

    The **Save** dialog box appears.

11. Type in the file name if you want to change it.
12. Browse to the folder where you want to save the file.
13. Click **Save**.
14. In the **Data Streams** dialog box, click **OK**.

#### Editing

To edit a PPP TCP/IP data stream:

1. On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Data Streams.**

   The **Data Streams** dialog box appears.

2. Select stream that you want to edit.
3. Click **Edit Stream**.

   The **PPP TCP/IP Data Stream** dialog box appears.

4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:

   | | |
   |---|---|
   | **PPP Header** | See **Section 2.11.21.11**. |
   | **TCP Header** | See Section 2.11.21.10. |
   | **IP Header** | See Section 2.11.21.9. |
   | **Data Payload** | See Section 2.11.21.12. |

6. Click **Delete Frame** to delete the selected frame.
7. Click the  and  arrows to change the order of the frames.
8. Click **OK** when done.

### 2.11.21.6 Creating an IP Packet Pool

In the ATM data stream, you can create IP Packet Pools. Do the following:

1. Create an ATM data stream.
2. Create a PDU.
3. In the Create AAL5 PDU dialog box, click **Multiple packets from pool** in the lower-left corner.
4. Click **Create Pool**.

   The **IP Packet Pool** dialog box appears.

5. Click **Create Packet(s)**.

   The **IP Packet dialog** box appears.

6. Click **IP Header** (see Section 2.11.21.9).
7. Click **Payload** (see Section 2.11.21.12).

   You can specify a fixed packet size or, if you are creating multiple packets, a size that is randomly selected from within a specified range or that is incremented within a specified range.
8. Specify the number of packets you want to create
9. Click **Create**.

The created packets are added to the pool. The dialog box remains active so you can change settings and create additional packets.

10. When you are finished creating packets, click **Close**.

To delete a packet:

1. Click the packet you want to delete.
2. Click **Delete Packet**.

To edit a packet:

1. Click the packet you want to edit.
2. Click **Edit Packet**.

To change the order of the packets:

1. Click a packet.
2. Click the up or down arrow buttons to move the frame up or down in the list.

When you are finished creating the packet pool:

1. Click **OK**.

   The **Specify File...** dialog box appears.
2. Browse to the folder where you want to store the file.
3. Type the name of the file in the **File Name** box.
4. Click **Save**.

The packet pool that you just created appears in the **Select packet pool...** box.

5. Click on the name of the pool.
6. Click **Create**.
7. If done, click **Close**.

   The **ATM Stream** dialog box appears.
8. Click **OK**.

### 2.11.21.7 Specifying a Custom Header

1. Create a Custom Ethernet IP data stream (see Section 2.11.21.2).
2. In the **Custom Ethernet IP Data Stream** dialog box, click **Create Frames**.

3. Click **Custom Header**.
4. Enter values directly into the box to the right.

### 2.11.21.8 Specifying an Ethernet Header

1. Create a Custom Ethernet IP data stream (see Section 2.11.21.2), an Ethernet IP data stream (see Section 2.11.21.3), or an Ethernet TCP/IP data stream (see Section 2.11.21.4).
2. In the **Ethernet IP Data Stream** dialog box, click **Create Frames**.
3. Click **Ethernet Header**.
4. Enter values directly into the boxes.

   If you are creating multiple frames and want each frame to have a different value for the destination or source MAC address, click **Advanced** next to the appropriate address box.

   The **Specify how you want...** dialog box appears. It displays options for generating MAC addresses.

   - Click **Fixed** if you want all frames to have the same address.
   - Click **From within range** if you want addresses chosen from a range that you specify.
     - — Click **Sequential** to have addresses chosen sequentially from within the range, starting within the range's lower bound.
     - — Click **Random** for random selection of addresses from within the range.
   - Click **From list** if you want addresses chosen from a list that you specify.
     - — Click **Sequential** to have addresses chosen sequentially from the list, starting within the first address in the list.
     - — Click **Random** for random selection of addresses from the list.
     - — To add an address to the list either click the  button or double-click beneath the last address in the list, enter the address value and pressing ENTER.
     - — To delete an address from the list, select it, then click the  button.
     - — To move an address up or down in the list, select it and click the  or the  button.

The list of addresses is saved in the Windows registry, so it is available during future Workbench sessions.

### 2.11.21.9 Specifying an IP Header

1. Create a Custom Ethernet IP data stream (see Section 2.11.21.2), an Ethernet IP data stream (see Section 2.11.21.3) or an Ethernet TCP/IP data stream (see Section 2.11.21.4).
2. In the **Ethernet IP Data Stream** dialog box, click **Create Frames**.

   The **Create frame(s)** dialog box appears.
3. Click **IP Header**.
4. Enter values directly into the boxes.
   - If you want the packet length to be automatically computed based on the length of the encapsulated payload, select **Computed** next to the **Packet length** box. Otherwise, the value you enter will be used without modification.

- If you want the header checksum to be automatically computed, select **Computed** next to the **Header checksum** box. Otherwise, the value you enter will be used without modification.
- If you are creating multiple frames and want each frame to have a different value for the **Source IP address** or **Destination IP address**, click **Advanced** next to the corresponding address box.

  The **Specify how you want...** dialog appears.

  In the **Frame size (in bytes)** area:

  — Click **Single** if you want all frames to have the same address then specify the address.

  — Click **From within range** if you want addresses chosen from a range that you specify. To have addresses chosen sequentially from within the range, starting within the range's lower bound, click **Sequential**. For random selection of addresses from within the range, click **Random**.

  — Click **From list** if you want addresses chosen from a list that you specify.


  – To add an address to the list, enter the address in the box to the right of the list then click **Add Address**.

  – To delete an address from the list, select it then click the  button.

  – To move an address up or down in the list, select it and click the  or the  button.

  – To have addresses chosen sequentially from the list, starting within the first address in the list, click **Sequential**.

  – To have addresses chosen randomly from the list, click **Random**.

## 2.11.21.10 Specifying a TCP Header

1. Create an Ethernet TCP/IP data stream (see Section 2.11.21.4) or an PPP TCP/IP data stream (see Section 2.11.21.5).
2. In the **Ethernet IP Data Stream** dialog box, click **Create Frames**.
3. Click **TCP Header**.
4. Enter values directly into the boxes.
5. If you want the checksum to be automatically computed, select **Computed** next to the **Checksum** box. Otherwise, the value you enter will be used without modification.

## 2.11.21.11 Specifying a PPP Header

1. Create a PPP TCP/IP data stream (see Section 2.11.21.5).
2. In the **PPP TCP/IP Data Stream** dialog box, click **Create Frames**.
3. Click **PPP Header**.
4. Enter the protocol value in the box to the right.
5. Choose between 8-bit and 16-bit protocol in the same box to the right.

### 2.11.21.12 Specifying a Data Payload

1. Create or edit a data stream of any type containing frames (any except an ATM data stream).
2. Click the **Data Payload** button to display the options for specifying the data payload for a frame.
3. Select a pattern from the **Fill pattern** list.
4. If you select an incrementing or decrementing pattern, you can specify the starting value for the fill operation in the **Hex starting value** box.
5. If you are creating multiple new frames you also have the option of having the incrementing or decrementing span the set of frames being created. For example, if the first frame is created with data 00 01 02... 4f, the second frame will have data 50 51 52..., and so on.
6. If you are editing an existing frame, you can choose to edit the data directly by clicking **Custom Data**, then editing the data fields within the box.

### 2.11.21.13 Specifying Frame Size

Specify a fixed frame size or, if you are creating multiple frames, a size that is randomly or incrementally selected from within a specified range.

To do this:

1. Create or edit a data stream of any type containing frames.
2. In the **Create Frame(s)** dialog box, go to the **Frame size (in bytes)** area and do one of the following:
   - Click **Fixed** and type the frame size in the **Fixed** box.
   - Click **Random** and type the **from** and **to** values in the boxes to the right.
   - Click **Increment** and type the **from** and **to** values in the boxes to the right.

### 2.11.21.14 Assigning I/O to Device Ports

After you have configured the IX Bus with devices and ports and created or imported data streams, you need to specify the input and output (I/O) for each enabled port. A port is connected to the IXP12nn on one side and to the ‘network’ on the other. Data comes into the port from the network and is placed into the receive buffer. Conversely, data is taken from the transmit buffer and sent out to the network.

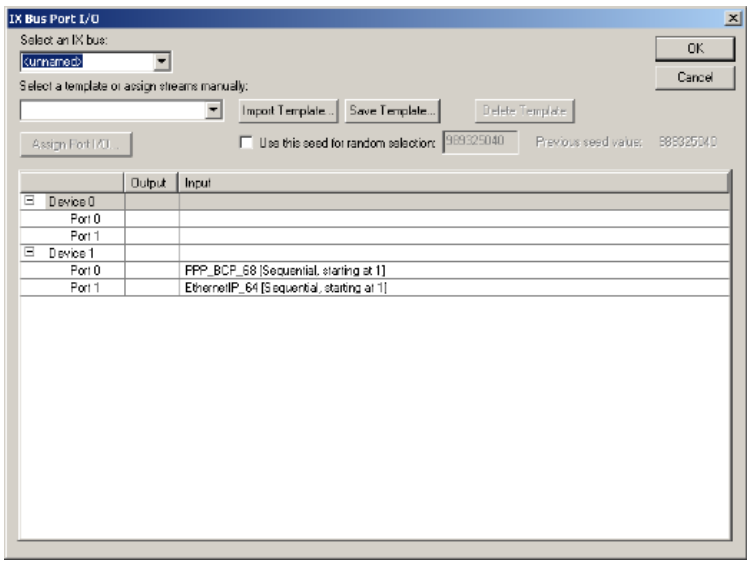
To effectively simulate a port’s operation, network traffic must also be simulated. Input from the network can either be simulated by the Workbench using data from streams or you can provide a DLL that supplies the input data. Output to the network can be thrown away or you can provide a DLL that receives the transmitted data. For details on how to implement a network traffic simulation DLL, see Section 2.11.22.

To assign I/O to ports:

- On the **Simulation** menu, click **IX Bus Device Simulator**, then click **Port I/O**.

  The **IX Bus Port I/O** dialog box appears. (See Figure 2-10.)

**Figure 2-10. The IX Bus Port I/O Dialog Box**




If there is more than one IX Bus in your system configuration, select the **IX Bus** to which you want to assign streams.

### Information Window

**Column one** — Lists the devices and ports for the selected IX Bus.

**Column two** — Shows the name of the DLL that is assigned to receive output from the port. If no DLL is assigned, the column is blank.

**Column three** — Shows either the name of the DLL that is to supply input data to the port or the data streams assigned to supply input data. In the latter case, the method by which packets are selected from the streams is shown in brackets after the stream names. If no input is assigned, the column is blank.

To change the I/O assignments for a port:

1. Click the port.
2. Click **Assign Port I/O**.

   The **Assign IX Bus Port I/O** dialog box appears.

### The Assign IX Bus Port I/o Dialog Box

In the **Transmit** area at the top of the **Assign IX Bus Port I/O** dialog box, select how you want to process data taken out of the port's transmit buffer. Select:

**No output** — If you don't want to process it, or

**Send data to DLL** — If you want the data passed to your network simulation DLL. Enter the full file path for the DLL or click the [...] button to browse to and select the DLL.

In the **Receive** area at the bottom of the dialog box, select how you want to supply input data for the port's receive buffer. Select:

**No input** If you don't want the port to receive any data, or

**Get data from DLL** If you want the Workbench to get data from your network simulation DLL. Type the full file path for the DLL or click the ... button to browse to and select the DLL, or

**Get data from streams** If you want the Workbench to get data from data streams.

- All data streams associated with the project along with their type are displayed in the list labeled List of all data streams.
- The streams that are assigned to the port are displayed in the list box labeled Assigned streams.

To assign a stream to the port:

1. Select the stream in the **List of all data streams** list.
2. Click **Assign**.

   The selected stream is added at the end of the assigned streams list. If a stream is already assigned, it is not assigned again.

To deassign a stream:

1. Select the stream in the **Assigned streams** list.
2. Click **Remove**.

In the **How to select...** area at the bottom right of the dialog box are controls that allow you to specify the method by which packets are selected from the assigned streams to be received by the port. When multiple streams are assigned, the Workbench treats them as one continuous sequence of packets.

Click **Sequential, Starting at packet**
If you want the packets to be selected sequentially from the stream(s). You can specify which packet you want to be the first packet received by the port. After the port receives the last packet in the last assigned stream, the Workbench wraps to the first packet in the first assigned stream.

Click **Random**
If you want packets to be selected at random from the assigned streams. For ATM streams, the PDUs are selected at random but the ordering of cells within the PDU is always maintained. For example, assume a stream has five PDUs. If PDU#2 is selected, its first cell will then be placed into the receive buffer. The next time PDU#2 is selected, its second cell is placed in the buffer, and so on, until all cells in the PDU are selected. Then the first cell is selected again.

Click **Interleaved, starting at packet**
If you want interleaved cell selection for ATM streams. PDUs are selected sequentially but only one cell in a PDU is selected at a time. For example, assume an ATM stream has three PDUs, with PDU#1 having one cell, PDU#2 having three cells and PDU#3 having two cells.

The packet selection sequence will be:

PDU#1 Cell#1

PDU#2 Cell#1
PDU#3 Cell#1
PDU#1 Cell#1
PDU#2 Cell#2
PDU#3 Cell#2
PDU#1 Cell#1
PDU#2 Cell#3
PDU#3 Cell#1

For non-ATM streams, the sequential and interleaved choices are identical.

When you have completed assigning streams and specifying the packet selection method, click **OK** to apply your choices and close the **Assign IX Bus Port I/O** dialog box.

### Random Selection Seeds

In the IX Bus Port I/O dialog box, you can force the random selection generator to use the number used on the last simulation, or you can type a new number. The number displayed is the number last used. To keep the same seed value, select **Use this seed for random selection**.

### Templates

You can perform the I/O assignment by selecting a template from the **Select a template...** list. The ports I/O is assigned based on the configuration specified in the template. The number of devices and number of ports in each device on the selected IX Bus must match the values in the template.

***Note:*** All streams that are assigned in the template must be in the project.

You can create your own template by assigning I/O then:

1. Click **Save Template**.

   The **Save Template** dialog box appears.

2. Type the name for your template and specify a file path where the template will be saved.

   You can browse to a file or folder by clicking the ... button. The default file extension for template files is .wbt.

If you have template files that you copied from other sources, you can import them into your Workbench configuration.

To do this:

1. Click **Import Template**.

   The **Import Port I/O Template** dialog box appears.

2. Browse to the file(s) you want to import.
3. Select one or more files and click **OK**.

   The files that you import appear in the **Select a template...** list.

4. Select the template from this list.

   When you have completed assigning I/O to each port:

5. Click **OK** to apply your assignments and dismiss the **IX Bus Port I/O** dialog box.

## 2.11.22 About Network Traffic Simulation DLLs

To simulate receiving or sending packet data from or to the network you can provide a dynamic-link library (DLL). You assign this DLL to the input and/or output side of an IX bus device port (see Section 2.11.21.14).

A network traffic simulation DLL must provide the following functions:

```
Initialize
Close
```

If the DLL is assigned to supply data to a port, then it must also have the following functions:

```
InitializeRxPortEx
InitializeRxPort
GetNextByte
GetInterpacketTime
GetReturnStatus
CloseRxPort
```

If the DLL is assigned to take data from a port, then it must also have the following functions:

```
InitializeTxPortEx
InitializeTxPort
SendNextByte
CloseTxPort
```

When you Start Debugging, the Workbench loads the DLL and calls the `Initialize()` function immediately after the Transactor initializes the model. This is the only time that this function is called. In the `Initialize()` function you can register console functions with the Transactor. You can then call these functions from a script in order to configure your traffic simulation. To call the Transactor you must include the file xact_dll.h and link against xact_dll.lib.

When you activate the Workbench to start receiving packets, it calls the DLL to initialize each port. If you have checked the option for packets to be received at the first clock cycle (see Section 2.11.23.1), then the initialization occurs at the first clock cycle. Otherwise, it will occur when the `start_packet_receive()` function is invoked from your startup script. In either case, the Workbench iterates through all the ports on every IX bus device.

If the DLL is connected to the receive side of the port, then the function `InitializeRxPortEx()` is called if you have provided it. Otherwise, the function `InitializeRxPort()` is called.

If the DLL is connected to the transmit side, then the function `InitializeTxPortEx()` is called if you have provided it. Otherwise, the function `InitializeTxPort()` is called.

***Note:*** See the file `NetTrafficSimInterface.h` for the contents of the port configuration data structures that are passed to these initialization functions.

As the simulation progresses, the Workbench calls function `GetNextByte()` whenever a byte of data is required on a receive port. In addition to the byte of data, the DLL must also return a flag indicating whether the byte is the last byte (EOP) in the frame/cell. When the Workbench receives the last byte, it will call the `GetReturnStatus()` function if you have provided one. It also calls the `GetInterpacketTime()` function in order to determine how long it waits before asking for the first byte of the next frame/cell. This allows the DLL to randomize the arrival of frames/cells.

On the transmit side, the Workbench calls the function `SendNextByte()` when it takes a byte of data is out of the transmit buffer to be sent over the network. An EOP flag is asserted along with the last byte in the frame/cell.

When the user Stops Debugging, the functions `CloseRxPort()` and `CloseTxPort()` are called for each connected receive and transmit port, respectively.

When the user closes the project or exits the Workbench, the function `Close()` is called just before the DLL is freed.

A Visual C++ project that is an example of a network traffic simulation DLL can be found at:

`...\IXP1200\WIN32DevTools\Samples\NetworkTraffic.`

## 2.11.23 About IX Bus Device Simulation Options

The Workbench provides several options that determine how the IX Bus device simulation behaves.

To do this:

- On the **Simulation** menu, click IX **Bus Device Simulator**, then click **Options**.

  The **IX Bus Device Simulation Options** dialog box appears. It has three tabs:

  | | |
  |---|---|
  | **Options** | See Section 2.11.23.1. |
  | **Logging** | See Section 2.11.23.2. |
  | **Stop Control** | See Section 2.11.23.3. |

### 2.11.23.1 About Miscellaneous Options

In the **IX Bus Device Simulation Options** dialog box:

- On the **Simulation** menu, click IX **Bus Device Simulator**, then click **Options**.
- Select **Automatically start receiving packets at first clock cycle** to specify that the ports start receiving packets from their assigned data streams at the first IX Bus clock cycle.

  If this option is cleared, then packets won't be received until you execute the command line function `start_packet_receive()`. This can be done in several ways:

  — Go to the **Command Line** window and type the command *start_packet_receive();*, or

  — Add the command `start_packet_receive();` to one of your startup scripts at the point where you want packet reception to begin, or

  — Create a command script (see Section 2.11.6) with the command *start_packet_receive();*, add the command script's button to the toolbar, then click the button when you want to start receiving packets.

- Enable **Run unbounded (infinite wire speed)** to have data always ready to be received by the IXP12nn and to have the ports always ready to receive data from the IXP12nn.

  This makes the simulation acts as if data was coming from and going to the network at infinite speed, bypassing the receive and transmit buffer and ignoring the data rate and interpacket gap values set for the port. In this mode, RxRdy and TxRdy are always asserted for every port.

  If this option is cleared, data is received from and transmitted to the network at the specified data rate with an interpacket gap.

- Enable **Stop simulation if a receive overflow occurs** to control whether or not the Workbench stops the simulation when a receive overflow occurs.

- Select or clear **Stop simulation if a transmit underflow occurs** to control whether or not the Workbench stops the simulation when a transmit underflow occurs.

The **IX Bus Device Status** window (see Section 2.11.23.4) displays receive and transmit rate data.

- In the **IX Bus Device Simulation Options** dialog box, select the units in which you want the rates displayed:

  **Megabits per seconds (Mbps) at the network interface**—this calculation includes pruned bits and bits that would have been processed in an interpacket gap.

  **Megabits per seconds (Mbps) at IX Bus**.

  **Frames per second (fps)**.

  **Cycles per frame**—this represents the average number of cycles between.

For ATM streams, a cell is considered to be a frame. For each port, the Workbench starts counting cycles for the receive rate calculation when the port asserts start-of-packet (SOP) for the first packet received by the IXP12nn after the user starts debugging or reset statistics. For the transmit rate calculation, cycle counting starts when the port gets start-of-packet (SOP) for the first packet transmitted by the IXP12nn after the user starts debugging or reset statistics.

- By default, the Workbench updates the data displayed in the **IX Bus Device Status** window only when simulation stops. However, when you select **Update status window every *xxx* IX bus cycles**, the Workbench updates the status at the specified interval while the simulation is running.

When you have completed specifying all your options, click **OK** to apply your choices and dismiss the dialog box or select another tab.

### 2.11.23.2 About Logging

In the **IX Bus Device Simulation Options** dialog box, click the **Logging** tab in order to specify whether and how the logging of received and transmitted packets is to occur. Logging is done on a per-port basis with receive and transmit logs being written to separate files. Only complete packets are logged. This means that if you enable logging during simulation, logging for a port will start when the next SOP occurs. Similarly, if you disable logging, any packet that is in the process of being received or transmitted will not get logged. You can clear all log files at any time by clicking Clear log files. Existing log files are automatically cleared when the first packet is logged after debugging is started.

- Select or clear **Enable Logging** to toggle whether packet logging occurs or not. This is a global setting that determines whether the individual port logging settings are in effect.
- Select or clear **Log frame numbers** to toggle whether or not frame numbers are logged along with the packet data. If enabled, the frame number appears as the first item on a line. Frame numbering starts when debugging is started, with the first packet received on a port and first packet transmitted to a port being number 1. The numbers continue to increment regardless of whether logging is enabled or not. So if you enable logging, disable it, then enable it again you will see a gap in the logged frame numbers.
- Select or clear **Log IX bus cycle times for SOP and EOP** to toggle whether or not to log the cycle times at which the first byte and last byte of a packet are received or transmitted. If enabled, the cycles times appear before the packet data on the line but after the frame number, if **Log frame numbers** is also selected.
- If logging of both frame numbers and cycle times are enabled, the logged data looks like:

```
25 4387 4395 01010101010202020202...
```

- If you want to get a log of packets received by the IXP12nn to the port, select **Log packets received by IXP from this port to file:** and type a file path in the box. You can browse to a file by clicking the ... button to the right of the box. The packet data is written to the file in hexadecimal format with one packet per line.
- If you want to get a log of packets transmitted by the IXP12nn to the port, select **Log packets sent by IXP to this port to file:** and type a file path in the box. You can browse to a file by clicking the ... button to the right of the box. The packet data is written to the file in hexadecimal format with one packet per line.
- If there is more than one IX Bus in your system configuration, select the IX Bus for which you want to specify logging. Select a port and specify whether or not you want received packets and/or transmitted packets logged.

When you have completed specifying all your options, click **OK** to apply your choices and dismiss the dialog box or select another tab.

### 2.11.23.3 About Stop Control

In the **IX Bus Device Simulation Options** dialog box, click the **Stop Control** tab in order to specify whether and when you want simulation or packet reception to stop.

If there is more than one IX Bus in your system configuration, select the IX Bus for which you want to specify stop control.

There are two bus-specific options:

- If you want to stop the simulation after a specific number of packets are received by the IXP12nn from the selected bus, select **Stop the simulation after the next *nnn* packets are received by the IXP from this bus** and type the number of packets in the box.

  When the specified number of packets are received, the simulation stops and a message box is displayed. If you continue the simulation from that point, it will again stop after the next nnn packets are received.
- If you want to stop the simulation after a specific number of packets are transmitted by the IXP12nn to the selected bus, select **Stop the simulation after the next *nnn* packets are transmitted by the IXP to this bus** and type the number of packets in the box.

To specify port-specific options, select a port from the list of devices and ports. The options are:

- If you want to enable the port to receive packets from the network, select **Send packets to the IXP from this port**. Ports are always enabled to accept packets transmitted by the IXP12nn.
- If you want to take action after a specified number of packets are received by the IXP12nn from the port, select **After the IXP receives the next *nnn* packets from this port:**, type the number of packets in the box, then click **Stop sending packets from this port to the IXP** or click **Stop the simulation**.
- If you want to take action after a specified number of packets are transmitted from the IXP12nn to the port, select **After the IXP transmits the next *nnn* packets to this port:**, type the number of packets in the box, then click **Stop receiving packets from the IXP on this port** or click **Stop the simulation**.

When you have completed specifying all your options, click **OK** to apply your choices and dismiss the dialog box or select another tab.

### 2.11.23.4 About the IX Bus Device Status Window

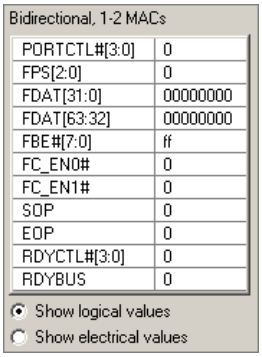
After you start debugging, you can view the status of the IX Bus device simulation in the IX Bus Device Status window.

To toggle visibility of the **IX Bus Device Status** window:

- On the **View** menu, click **Debug Windows**, then click **IX Bus Device Status**, or
  Click the  button on the **View** toolbar.

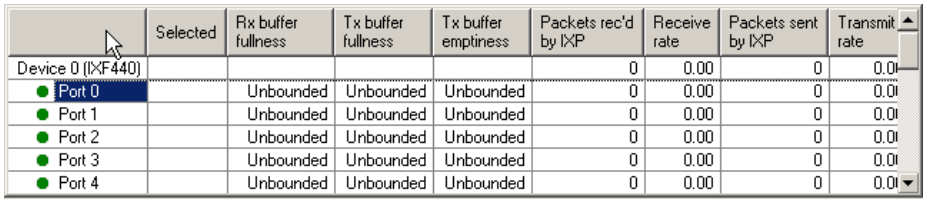
If there is more than one IX Bus in your system configuration, select the IX Bus for which you want to view status.

The status comprises:




- The mode that the IX Bus is operating under: unidirectional or bidirectional, and 1-2 MACs or 3+ MACs.
- The state of the IX Bus signals. Since some of the signals are asserted low (those whose names end in #), you select whether to display electrical values (0 = low, 1 = high) or logical values (0 = not asserted, 1 = asserted).
- The total number of packets received by the IXP12nn from this bus and the rate at which they were received in minpackets per second (a minpacket = 64 bytes).
- The total number of packets transmitted by the IXP12nn to this bus and the rate at which they were transmitted in minpackets per second.


- Important per-device and per-port data:



| | Selected | Rx buffer fullness | Tx buffer fullness | Tx buffer emptiness | Packets rec'd by IXP | Receive rate | Packets sent by IXP | Transmit rate |
|---|---|---|---|---|---|---|---|---|
| Device 0 (IXF440) | | | | | 0 | 0.00 | 0 | 0.0 |
| Port 0 | | Unbounded | Unbounded | Unbounded | 0 | 0.00 | 0 | 0.0 |
| Port 1 | | Unbounded | Unbounded | Unbounded | 0 | 0.00 | 0 | 0.0 |
| Port 2 | | Unbounded | Unbounded | Unbounded | 0 | 0.00 | 0 | 0.0 |
| Port 3 | | Unbounded | Unbounded | Unbounded | 0 | 0.00 | 0 | 0.0 |
| Port 4 | | Unbounded | Unbounded | Unbounded | 0 | 0.00 | 0 | 0.0 |

  — An indication as to whether a port is enabled to send packets to the IXP12nn. If it is, a green circle is displayed before the port identifier. It if isn't, a red circle is displayed.

  — An indication as to whether a port is receive- or transmit-selected by the IXP12nn. If it is receive-selected, the Selected column with display 'Rx'. If it is transmit-selected a 'Tx' is displayed.

  — The Rx buffer fullness–the number of bytes in the Rx buffer. If the number is equal to or greater than the RxRdy threshold or if the buffer contains at least on end-of-packet byte, then a green square is displayed next to the byte count indicating that the port will assert its receive ready bit.

  — The Tx buffer fullness–the number of bytes in the Tx buffer. If the number is equal to or greater than the TxSend threshold or if the buffer contains at least on end-of-packet byte,

then a green square is displayed next to the byte count indicating that the port will send Tx buffer data to the network.

— The Tx buffer emptiness–the number of free bytes in the Tx buffer. If the number is equal to or greater than the TxRdy threshold, then a green square is displayed next to the byte count indicating that the port will assert its transmit ready bit.

— The total number of packets received and transmitted by the IXP12nn from and to each device and the receive and transmit rates for the device.

— The total number of packets received and transmitted by the IXP12nn from and to each port and the receive and transmit rates for the port.

If you have specified that the simulation run unbounded, then the receive and transmit buffers are bypassed. So the word Unbounded is displayed instead of values for Rx buffer fullness, Tx buffer fullness and Tx buffer emptiness.

- Detailed data for a selected port. Select the port for which you want to view detailed data by clicking on it in the summary data list. The detailed data comprises:

  — The RxRdy threshold and the Rx buffer size.

  — The Rx buffer contents in hexadecimal format. A '**' entry indicates end-of-packet.

  — The number of EOPs in the Tx buffer.

  — The TxRdy and TxSend thresholds and the Tx buffer size.

  — The Tx buffer contents in hexadecimal format. A '**' entry indicates end-of-packet.

## 2.11.24 About Debug Configuration

The Workbench supports debugging in four different configurations:

| Mode | Foreign Model | Comments | Reference |
|---|---|---|---|
| Local Simulation | None | The Workbench and the IXP12nn Network Processor simulator (Transactor) both run on the Windows platform. | Section 2.11.24.1 |
| Local Simulation | Local | The Workbench, the Transactor and a foreign model Dynamic-Link Library all run on the same Windows platform. | Section 2.11.24.2 |
| Local Simulation | Remote | The Workbench and the Transactor both run on the same Windows platform and communicate over the network with a foreign model running on a remote system. | Section 2.11.24.3 |
| Hardware | None | The Workbench runs on a Windows host and communicates over a network or a serial port with a subsystem containing an actual IXP12nn Network Processor. | Section 2.11.24.5 |

### 2.11.24.1 Setting Up Local Simulation with No Foreign Model

To specify local simulation with no foreign model:

1. On the **Debug** menu, select **Simulation**, if not already checked.
2. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.
3. Click the **Foreign Model** tab.

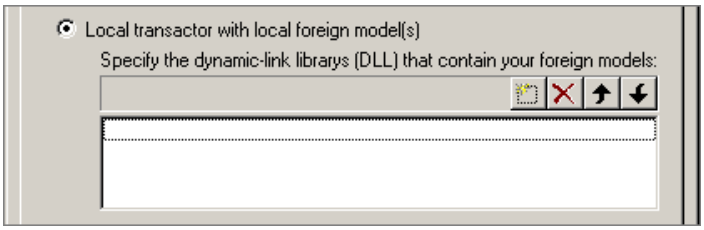
4. Click **Local transactor with no foreign model**.
5. Click **OK**.

### 2.11.24.2 Setting Up Local Simulation with Local Foreign Model

To specify local simulation with a local foreign model:

1. On the **Debug** menu, select **Simulation**, if not already checked.
2. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.




3. Click the **Foreign Model** tab.
4. Click **Local transactor with local foreign model(s)**.
   - Type, or browse for using the button, the complete file path for DLL that contains your foreign model.
   - Delete DLLs from your project using the button.
   - Move DLLs up or down in your project using the or the button.

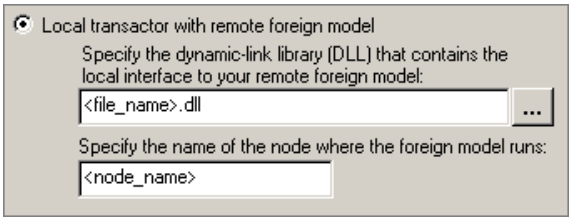
   (See Chapter 6 for details on creating a foreign model simulation extension).
5. Click **OK** when done.

### 2.11.24.3 Setting Up Local Simulation with Remote Foreign Model

To specify local simulation with a remote foreign model:

1. Ensure that Portmapper is running, see Section 2.11.24.4.
2. On the **Debug** menu, select **Simulation**, if not already checked.
3. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.




4. Click the **Foreign Model** tab.
5. Click **Local transactor with remote foreign model**.
6. Type, or browse for using the ... button, the complete file path for DLL that contains the local interface to your remote foreign model.
7. Type the name of the node where the foreign model runs in the **Specify the name...** box.
8. Click **OK** when done.

### 2.11.24.4 Installing PortMapper

PortMapper is automatically installed as part of the IXA SDK installation process. To ensure PortMapper is installed and running:

1. On the Window's task bar, click **Start**, **Settings**, **Control Panel**.
2. Double-click the **Services** icon.
3. In the **Services** dialog box, scroll down to find the **IXP1200 PortMapper** service.
4. Ensure this service is started by selecting the status column.
5. If the PortMapper service isn't started, highlight IXP1200 PortMapper with the mouse.
6. Click **Start**.

### 2.11.24.5 Setting up Hardware Debug

To debug hardware, you must specify how to connect to the subsystem(s) containing the IXP12nn Network Processor:

1. Ensure that Portmapper is running, see Section 2.11.24.4.
2. On the **Debug** menu, select **Hardware** if not already enabled.
3. On the **Hardware** menu, click **Options**.

   The **Hardware Options** dialog box appears.
4. Click the **Connections** tab.
5. Select a chip from the Select a chip box (<unnamed> if only one chip).
6. Enable the type of connection to that chip by clicking on the appropriate button:

| | |
|---|---|
| **No Connection** | If you have multiple chips in your project, you can specify that one or more not be connected. However, at least one must be connected. |
| **Connect via PCI bus** | This allows you to connect to an IXP12nn that is on the PCI bus in your PC. |
| **Connect via serial port to VxWorks** | You must specify the name of the node where the hardware is located. |
| **Connect via Ethernet** | You must specify the name of the node where the hardware is located. |

## 2.12 Running in Batch Mode

#### Workbench Batch Files

A Workbench batch file is an ASCII text file.

The first line must contain the complete path for a Workbench project file, for example, c:\mydir\router.dwp.

The Workbench opens the specified project and performs a build operation.

The second line must contain the keyword hardware or simulation to specify the debug configuration.

The Workbench starts debugging in the specified configuration.

All subsequent lines are executed by the command line interface.

To have the Workbench exit when it completes executing the batch file, place an exit command as the last line in the batch file.

Here is an example of a batch file:

```
c:\mydir\router.dwp
simulation
@foobar.ind
go 100
exit
```

### Batch Mode

You can run the Workbench in batch mode by specifying a Workbench batch file preceded by an @ as a program argument when starting the Workbench.

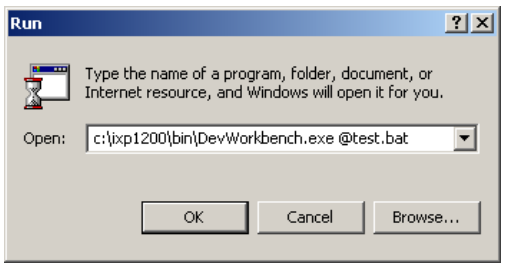
For example:

1. On the Windows task bar, click **Start**, and then click **Run**.

   The **Run** dialog box appears.

2. Type c:\ixp1200\bin\DevWorkbench.exe @test.bat in the **Open** box.

3. Click **OK**.




Windows launches the Workbench and executes the batch file test.bat.

## 2.13 An Exercise in Using the Workbench

The procedure that follows introduces you to basic Workbench functions by showing you how to create a project, insert some source microcode, build a project, and perform some simple debugging.

### Invoking The Workbench

From the **Start** button on the **Taskbar**:

- Click **Programs**, **Intel IXP1200**, **Developer WorkBench**.

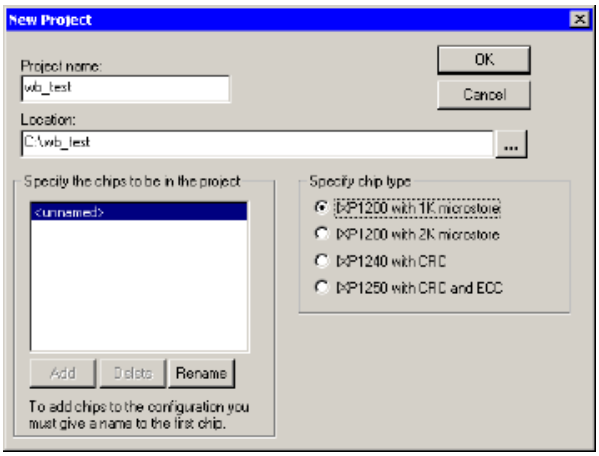
The selection IXP1200 is the default, but may be different if you assigned another name during the installation of your IXP1200 software. If the **Tip of the Day** appears, click **OK**.

Your screen should now display the Developer Workbench application (see Figure 2-1).

### Creating a New Project

1. On the **File** menu, click **New Project**.

The **New Project** dialog box appears.




2. Click the button to the right of the **Location** box, select C:\ and click **OK**.
3. Type *wb_test* in the **Project name** box.

   The text in the **Location** box should now display *C:\wb_test*. This is because the project name you typed actually becomes a folder containing two files—wb_test.dwp and wb_test.dwo. From this point on, all the project files and information will default to this folder or one of its subfolders.
4. From the **Select chip type...** list, select **IXP1200 with 1K microstore**.
5. Click **OK** when done.

   Note that your project, wb_test, appears in the **Project Workspace**. It contains a tree of supporting empty folders that will contain the various files needed to build the project.

### Creating a New Source File

1. On the **File** menu, click **New**.

   The **New** dialog box appears.
2. Select **Source File** from the list and click **OK**.

   This opens an empty document window (the background changes from dark gray to white). Note that the name *Source1* appears on the title bar.
3. Type the following two lines into the document:

```
#include "wb_gpr_ctx.uc"
#include "wb_gpr_abs.uc"
```

   Note that the name in the title bar changes from *Source1* to *Source1** because you modified the document. Also, the two #include entries are blue indicating that they are keywords.

### Saving the New Source File

1. On the **File** menu, click **Save As**.
2. Enter *wb_test* in the **File name** box.
3. From the **Save in** list, browse to folder *C:\wb_test* and click **Save**.

   The Workbench saves the file as wb_test.uc (microcode file).

### Inserting New Source File Into Project

1. On the **Project** menu, click **Insert Assembler Source Files**.

   The **Insert Assembler Source Files into Project** dialog box appears. The dialog box displays the folder (C:\) where you saved the file wb_test.uc.

2. Select the file wb_test.uc and click **Insert**.

   This inserts wb_test.uc into the project. Ignore, for now, the messages in the **Output** window about not finding files, if there is one. Note that in the **Project Workspace**, the folder labeled **Assembler Source Files** has a "+" to it's left. If you click it, the folder expands and the file name *wb_test.uc* appears.

*Note:* If you can't see the file wb_test.uc in the **Insert Assembler Source Files** dialog box, type c:\wb_test.uc in the **File name** box and click **Insert**. This causes the program to go directly to the file where you saved it.

### Selecting Build Settings

1. On the **Build** menu, click **Settings**.

   The **Build Settings** dialog box appears.

2. Click the **Assembler** tab.

3. Click **New** to add a .list output file to the project.

   The **Insert New List File into Project** dialog box appears.

4. Browse to C:\.

5. Type *wb_test.list* in the **File name** box and click **Insert List File**.

   The file wb_test.list appears in the **Output to target .list file** box. Also, in the **Path to target .list** file box just below, the full path of the .list file appears. The entry in this box should be C:\wb_test.list.

6. From the **Root File** list, select wb_test.uc (it should be the only choice).

7. Click the **Linker** tab.

8. From the list to the right of **MicroEngine 0**, select wb_test.list (it should be the only choice).

   This specifies that this file gets loaded into Microengine 0's microstore.

9. Click **OK**.

   Notice that the file wb_test.uc in the **Assembler Source Files** folder in the **Project Workspace** is now preceded by an R. This indicates that this file is a root file.

### Building the Project

1. On the **Build** menu, click **Build,** or

   This invokes the Assembler to assemble wb_test.uc. The Assembler results appear in the **Output** window at the bottom of the Workbench.

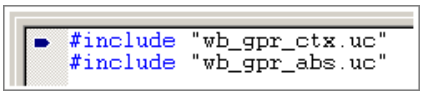
   Note that it found (2) errors and (0) warnings.

2. Scroll the **Output** window to view the errors if necessary.

3. Double-click the line containing the first error message.

   This highlights the error message and marks the source line on which the error occurs.

   In this case, the Assembler cannot open the two files that we included because they are not in the project folder. The next steps fix this problem.



4. On the **Build** menu, click **Settings** and then click the **General** tab.
5. In the **Assembler include directories** box, click the button.

   This activates an entry in the list box for editing.
6. Type *C:\IXP1200\MicroCode\workbench* into the entry or click the ... button to browse to this location.

   This step assumes you used the default folder name of *IXP1200* when you installed the IXP SDK software. This folder contains the microcode files that were included in the wb_test.uc file.

***Note:*** At this point you must press ENTER to accept the new folder. The ... button disappears.

7. Click **OK** to accept the changes and close the dialog box.
8. On the **Build** menu, select **Build**.
9. The build summary should now indicate (0) errors and (28) warnings, with the Linker producing a wb_test.uof file.

### Debugging

1. On the **Debug** menu, click **Start Debugging**. You can also press F12 or click the button.

   This initializes the simulation and enables all of the debug features of Workbench.
2. In the **Project Workspace**, click **ThreadView** ThreadView.
3. Expand the tree under the MicroEngine 0 entry by clicking the "+" to the left of it.
4. Double-click **Thread 0 (0)**.

   This opens a thread window for **Thread 0**. Click the button (if not already) to maximize this window.
5. Hide **Project Workspace** by clicking on the in its upper right corner or by clearing **Project Workspace** on the **View** menu. Similarly, hide the **Output** window (if present).
6. If you do not see the **Run Control** window, click **View**, Debug Windows, Run Control.

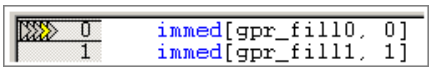
   The Run Control window appears (see Figure 2-1).
7. If you do not see the **Command Line** window, on the **View** menu, click **Debug Windows**, then click **Command Line**.

   The **Command Line** window appears (see Figure 2-1).

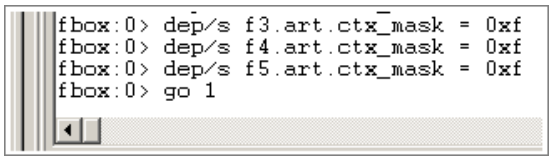
***Note:*** The **Run Control** and **Command Line** windows may appear automatically when you start debugging.

8. In the **Run Control** window, click the **Step** button.

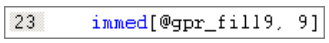
   This starts the Transactor and executes one cycle. The Pipe Stage marker should now point to the instruction that is in the P1 stage of the Microengine pipeline.



Notice that the **Command Line** output area logs the Transactor commands that are being executed.




9. Click **Step** four more times to see the Pipe Stage marker advance.
10. Using the mouse pointer, set the insertion point in the line that contains the instruction at line 23.

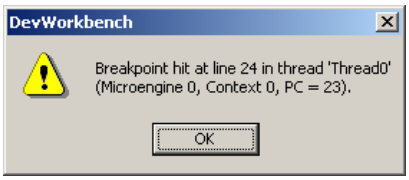



    Note: If you don't see the line numbers, right-click in the window and click **Display Instruction Addresses**.
11. Press F9 to set a breakpoint on that line.

    Pressing CTRL+F9 toggles whether the breakpoint is enabled or disabled . Make sure you leave the breakpoint enabled.
12. Type 200 into the box next to the **Go for** button.


13. Click the **Go for** button.

    The simulation stops at the breakpoint before completing the 200 cycles. The following message appears.




14. Click **OK**.

    The Pipe Stage marker should point to the line with the breakpoint (the breakpoint marker may be obscure under the Pipe Stage marker).
15. Click the **Go For** button again.

    The Pipe Stage marker should advance to the end of the thread window.


16. Move the cursor over the [gpr_fill 15,15] symbol in the thread window and leave it there for a second.

    A data tip appears showing the value in that register. In parenthesis, we also see to which bank the register was allocated and that the value is absolute.


17. Close the **Thread(0)** window (click the ☒ all the way to the right on the menu bar, or on the **Window** menu, click **Close**).
18. On the **File** menu, click **Exit**.

    This displays a dialog box asking you if you want to save changes to the project.
19. Click **Yes**.

### Restarting the Workbench

1. From the **Start** button on the Windows task bar, click **Start**, **Programs**, **Intel IXP1200**, **Developer WorkBench**.

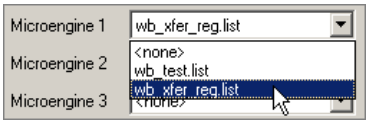
   If the **Tip of the Day** appears, click **OK**.
2. On the **File** menu, click **Recent Projects**, **C:\...wb_test**.
3. On the **Project** menu, click **Insert Assembler Source Files**.

4. Navigate to the folder C:\IXP1200\MicroCode\workbench (or wherever you may installed your kit).
5. Select the files wb_xfer_reg.uc, wb_sram_addr.uc, and wb_sdram_addr.uc.
6. Click **Insert**.
7. On the **Build** menu, click **Settings**, and then click the **Assembler** tab.
8. Click **New** to add a .list output file to the project.

   The **Insert New List File into Project** dialog box appears.
9. Type the file name *wb_xfer_reg* in the **File name** box and click **Insert List File**.

   The list file wb_xfer_reg.list appears in the **Output to target .list file** box.
10. From the **Root File** list, select the file wb_xfer_reg.uc.
11. Click the **Linker** tab.
12. From **Microengine 0** list, select **<none>** to exclude it from the linked image.
13. In the list to the right of **MicroEngine 1**, select wb_xfer_reg.list.

    This specifies that this file gets loaded into Microengine 1's microstore.




14. Click **OK**.
15. On the **Build** menu, click **Build** (or press F7).

    The message window displays:

    ```
    Build results: (1) errors, (0) warnings.
    ```
16. Press F4 to highlight the assembly error.

    A blue pointer points to the line `my_macro(reg_1)`. This indicates an error that occurred because of an incorrect source line that needs to be conditionally omitted from the assembly.
17. On the **Build** menu, click **Settings**, then click the **Assembler** tab.
18. Select wb_xfer_reg.list from the **Out to target .list file** list.
19. Type the symbol *`foobar`* into the **Preprocessor definitions** box.
20. Click **OK**.
21. On the **Build** menu, click **Build**.

This time the build should succeed with:

```
(0) errors, (4) warnings
```

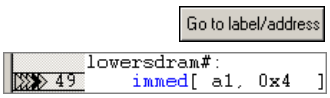
### Debugging the Warnings

1. On the **Debug** menu, click **Start Debugging** (or press F12, or click the  button.
2. On the **Window** menu, click **Close All**.
3. On the **View** menu, click **Project Workspace** (if not already checked).
4. Click **ThreadView** in the Project Workspace.
5. Expand the tree under the MicroEngine 1 entry.

6. Double-click **Thread 4 (1)**.
7. On the **View** menu, clear **Project Workspace** to hide the **Project Workspace** window.
8. In the **Run Control** window, click the … button to the right of the **Label(s)** box.

   You may need to expand the size of the Run Control window to see the button. Clicking the button displays the **Go To Labels** dialog box.
9. Select the label lowersram# and click **Add**.
10. Select the label lowersdram# and click **Add**.

    Notice that the two labels appear in the **Labels to go to** box.
11. Click **OK**.
12. Click **Go to label/address** in the **Run Control** dialog box.




    The simulation stops with thread 4 executing at the label lowersdram# (as displayed in the Command Line window).
13. Click **Step** 2 or 3 times.

    Notice that only thread 4's PC is advancing. This is because thread 5 is swapped out.
14. Click **Go to label/address** again.

    The simulation stops with thread 5 executing at the label lowersram#. Notice that thread 5 is now the active thread.
15. Click **Step** 2 times.

    Notice that only thread 5's PC is advancing.
16. Right-click the same line as the Pipe Stage marker in thread 4's window.
17. Select **Go To Source** on the shortcut menu.

    This opens the source file wb_sdram_addr.uc and places the cursor in the corresponding source line.
18. Close the wb_sdram_addr.uc source window (click the ☒ button).
19. Place the insertion cursor on the same line as the Pipe Stage marker in thread 5's window.
20. On the **Debug** menu, click **Go to source**.

    Notice the source file is different from the one displayed the last time you selected **Go To Source** for Thread 4.
21. Close the wb_sdram_addr.uc source window.
22. On the **Debug** menu, click **Run Control**, then click **Reset** (or press CTRL+SHIFT+F12).
23. In the **Run Control** window, click **Set Thread Filter**.


    The **Thread Filter For Go To** dialog box appears.
24. Clear all Microengines except **MicroEngine 1**.
25. Clear all four chip-wide contexts located across the top of the matrix of boxes.

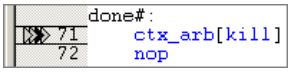
    This leaves the four **MicroEngine 1** context check boxes enabled.


26. Clear the check boxes for **MicroEngine 1** context 0, **MicroEngine 1** context 2, and **MicroEngine 1** context 3.

This leaves only MicroEngine 1 context 1 selected, which corresponds to thread 5 (out of the total of 24 threads in all six Microengines).


27. Click **OK**.
28. In the **Run Control** dialog box, click the ... button again to select a label.
29. Select the label done# and click **Add**.
30. Click **OK**.
31. Click **Go to label/address**.

    The simulation stops when thread 5 reaches the done# label.




### Terminating the Debug Session

Now that you have set the proper thread filters for this session, you can end the debugging.

- On the **Debug** menu, click **Stop Debugging** (or press CTRL+F12).

  The debugging session ends and you return to the main window.

### Rebuilding the Program

To check if the program runs properly:

- On the **Build** menu, click **Build** (or press F7).

The microcode should execute with no errors and no warnings, displaying the message

```
Build results: (0) errors, (0) warnings
```

### Exiting the Workbench

To exit the Developer Workbench:

1. On the **File** menu, click **Exit** (or press ALT+F4).

   This displays a dialog box asking you if you wish to save changes to the project (assuming you have made changes since it was last saved).

2. Click **Yes** if you want to save your changes.

# Assembler 3

This chapter provides information on running the Assembler. Background information on the Assembler functions appears in the *IXP1200 Network Processor Programmer's Reference Manual*.

## 3.1 Assembly Process

This section describes how to invoke the Assembler and the steps that it goes through in processing a microcode file.

### 3.1.1 Command Line Arguments

The Assembler is invoked from the command line:

```
uca [options] microcode_file microcode_file...
```

where the options consist of:

| | |
|---|---|
| -CPU=n | Use n=1 for the IXP12nn. |
| -m *file* | Loads the microword definitions from the specified file rather than the default. |
| -p *file* | Loads the ucc parse definitions from the specified file rather than the default. |
| -o *file* | Use *file* as the generated list file. This is only valid if there is one microcode_file. |
| -O | Enables optimization. |
| -g | Adds debugging info to output file. |
| -v | Prints the version number of the Assembler. |
| -h | Prints a usage message (same as -?). |
| -? | Prints a usage message (same as -h). |
| -new | Use new register allocation. |
| -t=n | Time-out in seconds for old register allocator. |
| -r | Register declarations are not required. |
| -R | Register declarations are required. |

The following revision arguments allow assembly to be targeted for a specific chip version or range of versions, overriding the default values. The predefined Preprocessor symbols __REVISION_MIN and __REVISION_MAX reflect the specified version range. In addition, the version range is also written to the .list file in a '.cpu_version' directive. Use the following:

| | |
|---|---|
| 0 | 1200 (A0) |
| 1 | 1200 (B0) |
| 2 | not used |
| 3 | 1200 (C0) |
| 4 | 1200 (D0) |

| | |
|---|---|
| 5 | 1240/1250 (A0) |
| 6 | 1240 (B0) |
| 7 | 1250 (C0) |

-REVISION=*n* Targets assembly to chip version *n*. This is equivalent to setting specifying options '-REVISION_MIN=*n*' and '-REVISION_MAX=*n*' with the same value.

-REVISION_MIN=*n* Targets assembly to the minimum chip version *n*. (the default is 0).

-REVISION_MAX=*n* Targets assembly to the maximum chip version *n*. (the default is 15, no limit).

The following options are passed to the preprocessor, UCP:

-P Preprocess only into a file: microcode_file.ucp.

-E Preprocess only into stdout.

-I*folder* Add the *folder* to the end of the list of directories to search for included files.

-D*name* Define *name* as if the contained "#define *name* 1"

-D*name=def* Define *name* as if the contained "#define *name def*"

-N Disable the pre-processor

The **microcode_file** names may contain an explicit suffix, or if the suffix is missing, .uc is assumed. Assembling several files in one command line is equivalent to assembling each individually; the files are not associated with each other in any way.

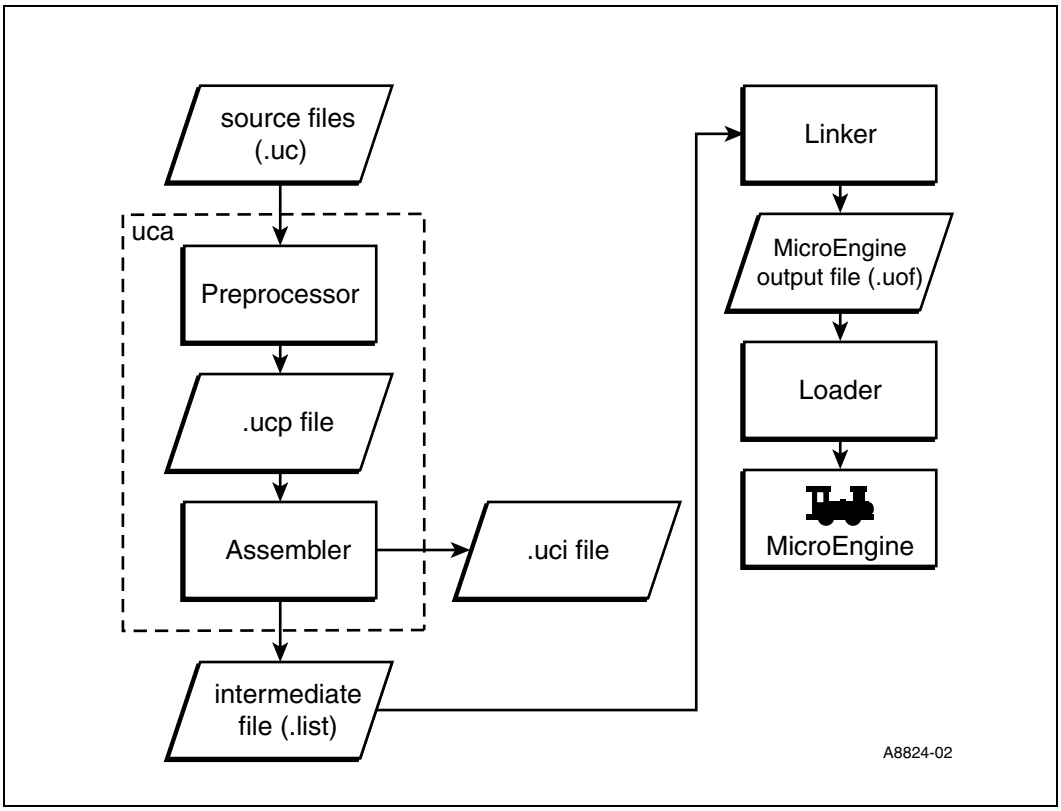
If uca is invoked with no command line arguments, then a usage summary is printed.

In Windows 95/NT environments, the Assembler may also be invoked through the IXP1200 workbench (see Section 2 for more information on the Developer Workbench).

## 3.1.2 Assembler Steps

As shown in Figure 3-1, invoking the Assembler results in a two-step process composed of a preprocessor step and an Assembler step. The preprocessor step takes a .uc file and creates a .ucp file for the Assembler. The Assembler takes a .ucp file and creates an intermediate file with the file name extension of .uci. The .uci file is used by the Assembler to create the .list file and provides error information that may be used in resolving semantic problems (such as register conflicts) in the input file.

**Figure 3-1. Assembly Process**




The .uc file contains three types of elements: microwords, directives, and comments. Microwords consist of an opcode and arguments and generate a microword in the .list file. Directives pass information either to the preprocessor, Assembler, or to downstream components (e.g., the Linker) and generally do not generate microwords. Comments are ignored in the assembly process.

The Assembler performs the following functions in converting the .uc file to a .list file:

- Checks microcode restrictions.
- Resolves symbolic register names to physical locations.
- Optimizes branches, in some cases by rearranging instructions near a branch to minimize branch penalties.
- Resolves label addresses.
- Translates symbolic opcodes into bit patterns.

## 3.1.3 Case Sensitivity

The microcode file is case insensitive, while the command line arguments are case sensitive.

### 3.1.4 Assembler Optimizations

If optimizations are enabled, instructions occurring before a branch may be moved after the branch to minimize branch latency effects. This optimization is illustrated by the following code segment example:

```
    alu[x,a,+,b]
    load_addr [link, nextline#1]
    br[sub1#]
nextline#:
    alu[y,a,-,b]
    nop
sub1#:
    immed[a,1]
    rtn[link]
```

The following instructions are assembled as a result:

```
    alu[x,a,+,b]
    br[sub1#], defer[1]
    ; branch latency fill optimization: the uword below
    ; was pushed down 1 position
    load_addr[link,nextline#]
nextline#:
    alu[y,a,-,b]
    nop
sub1#:
    rtn[link], defer[1]
    ;branch latency fill optimization: the uword below
    ;was pushed down 1 position
    immed [a,1]
```

### 3.1.5 Processor Revision

Over time, IXP1200 processors will be released in different versions with different features. There will be microcode written to use these features, but this microcode will fail if it is run on the wrong revision of the processor. To deal with this issue, the user can specify a range of revisions for which they want their microcode assembled. This is done using the -REVISION_MIN and -REVISION_MAX directives. For simplicity, the -REVISION directive can be used to set the min. and max. to the same value.

These directives have two effects: They define two preprocessor symbols: __REVISION_MIN and __REVISION_MAX. They also deposit the revision information into the output file to be used eventually by the loader.

The intent is that microcode that uses features that are available only on a range of processor revisions should check these predefined symbols to see if will work under all of the requested versions. If it does not, the microcode should signal an error via the #error directive. This is illustrated in the *IXP1200 Network Processor Programmer's Reference Manual*.

Thus, if the user requests that the generated code be valid for some range of processor revisions, and there are no errors during the assembly, then the user can assume that the generated code will in fact work on all processors in that range. If the user tries running it on a processor outside of that range, then the loader returns an error.

# Microengine C Compiler 4

The IXP1200 supports microcode compiled from C language code to support the Microengines and their threads. The StrongARM processor must be coded in assembly language and does not support the C language. You can create the C code using the DWB GUI or any available text editor. You can then compile and link the code using the GUI or the Compiler command line.

This chapter explains the subset of the C language supported by the IXP1200 Microengine C Compiler and the extensions to the language to support the unique features of the processor.

For information on the Compiler functions refer to the *Intel Microengine C Compiler Language Support Reference Manual*, PN 278426.

## 4.1 The Command Line

You can invoke the command line from a DOS window on your system. Do the following:

1. Open a DOS window.
2. Go to the folder containing the C Compiler files, typically:
   C:\IXP1200\bin>
3. Invoke the C Compiler using this command:
   uccl [options] filename [filename...]

## 4.2 Supported Compilations

Two kinds of compilations are supported:

- Compile a single source file (*.c, *.i) into one object (*.obj) file
- Compile any combinations of source file (*.c, *.i) and/or object file (*.obj) into one list file (*.list).

In the first case, you must use the -c switch in the command line.

***Example:*** uccl -c file1.c file2.i

In the second case, do not use the -c switch.

## 4.3 Supported Option Switches

Table 4-1 lists and defines all the supported C Compiler command line switches. The CLI warns and ignores unknown options. The CLI honors the last option if it conflicts with previous one, for example,

```
uccl -c -O1 -O2 file.c
```

generates the following warnings and proceeds:

```
uccl: Command line warning: overriding '-O1' with '-O2'
```

**Table 4-1. Supported CLI Option Switches (Sheet 1 of 2)**

| Switch | Definition |
|---|---|
| -?<br>-help | Lists all the available options. |
| -c | Compile each .c or .i file to a .obj file (rather than compile and link). |
| -On | Optimize for: n=1, size (default); n=2, speed; n=d, debug (turns off optimizations). |
| -Obn | Inlining control: n=0, none; n=1, explicit (inline functions declared with __inline or __forceinline (default)); n=2, any (inline functions based on Compiler heuristics, and those declared with __inline or __forceinline)" |
| -Qip_no_inlining | Turn off inter-procedural inlining, which is often too aggressive. |
| -Dname[=value] | Specify a #define symbol. The value, if omitted is 1. |
| -E<br>-EP<br>-P | Preprocess to stdout.<br>Preprocess to stdout, omitting #line directives.<br>Preprocess to file. |
| -Zi | Produce debug information. The Compiler generates a file with a .dbg extension. |
| -Qbigendian | Compile big-endian byte order (default). Compiler adds -DBIGENDIAN, -ULITTELENDIAN. All other command line BIGENDIAN/ LITTLEENDIAN symbol definitions and undefinitions are ignored. |
| -Qlittleendian | Compile little endian byte order. Compiler adds -DLITTLEENDIAN -UGIBIGENDIAN. All other command line LITTLEENDIAN/BIGENDIAN symbol definitions and undefinitions are ignored. |
| -Qperfinfo=n | Print performance information. n=0, no information (similar to not specifying); n=1, variables stayed in memory (not allocated in register) and the reason why. |
| -Fo<file> -Fo<Dir/> | Name of object file or directory for multiple files. |
| -Fe<file> | Base name of executable (.list, .ind) file. |
| -I path[;path2...] | Path(s) to include files, prepended before path(s) specified in environment variable UCC_INCLUDE. |
| -FI<file> | Force inclusion of file. |
| -Wn n=0, 1, 2, 3, 4 | Warning level.<br>0=print only errors<br>1, 2, 3=print only errors and warnings<br>4=print errors, warnings, and remarks. |

**Table 4-1. Supported CLI Option Switches (Sheet 2 of 2)**

| Switch | Definition |
|---|---|
| `-Qrevision_min=n`<br>`-Qrevision_max=m` | The version arguments allow the Compiler to generate code that will work on a range of processor versions.<br>ID Chip<br>0 1200 A0<br>1 1200 B0<br>2 not used<br>3 1200 C0<br>4 1200 D0<br>5 1240/1250 A0<br>6 1240 B0<br>7 1250 B0<br>Use min and max to set the range of processor versions that the code will run on.<br>Note: The loader will error if you try to load on an incorrect processor version. |
| `-Qnthreads=<1,2,3,4>` | Specify number of thread, default to 4. |
| `-Fa<filename>` | Produces a .uc file containing the generated microcode intermixed with the source program lines. |
| `-link` | Call Microengine image linker (ucld) after successful compilation, passing default options<br>"-u 0 -sr 4:0xfffffffc -sc 4:0x0ffc -dr 0x10:0xfffffff0" |

## 4.4 Compiler Steps

The .uc file contains three types of elements: microwords, directives, and comments. Microwords consist of an opcode and arguments and generate a microword in the .list file. Directives pass information either to the preprocessor, Assembler, or to downstream components (e.g., the Linker) and generally do not generate microwords. Comments are ignored in the assembly process.

The Assembler performs the following functions in converting the .uc file to a .list file:

- Checks microcode restrictions.
- Resolves symbolic register names to physical locations.
- Optimizes branches, in some cases by rearranging instructions near a branch to minimize branch penalties.
- Resolves label addresses.
- Translates symbolic opcodes into bit patterns.

## 4.5 Case Sensitivity

The C language code as well as the command line switches are case sensitive.

# Linker 5

The Linker is used to link microcode images. Microcode images are generated by the microcode Compiler or Assembler, whereas application objects are generated by a StrongARM C/C++ compiler. The method is C/C++ compiler independent. Two types of pointers are bound between microcode and StrongARM Core application objects:

- Shared address pointers
- Function pointers

This chapter describes how to use the microcode Linker, ucld. The task of ucld is to process one or more microcode Compiler or Assembler, (uca) output files,*.list, and create an object file that can be loaded by the microcode loader. The loader, UcLo, is a library of C functions that facilitate external address pointer resolution and the loading of images to the appropriate Microengine. The microcode loader is described in the *IXP1200 Network Processor Software Reference Manual* and uca is described in Chapter 6.

## 5.1 About the Linker

Memory is shared between the StrongARM Core and the Microengines. A common mode of design will have the StrongARM Core generate and maintain data structures, while the Microengine reads the data. Common address pointers will be used for these data structures. For example, the base address of a route table will need to be shared. The solution will allow microcode and StrongARM applications to be written and compiled that can access the common address pointer.

### 5.1.1 Configuration and Data Accessed by the Linker

Various Microengine configuration data structures will exist in the StrongARM Core. These are used to hold information about Microengines. Key repositories are:

- Microcode object. This is the binary object containing images that can be loaded into microcode.
- Microcode page/functionality map, per Microengine. Association of microcode image to Microengines. This is a list of libraries currently loaded. Separate list gives which libraries can be loaded (dynamically paged), but are currently in memory.
- Thread functionality map, per Microengine context. Each thread is identified as either a service thread or an autonomous thread. Each service thread has a library list of what libraries it supports.
- Shared address imports/externs list, per microcode image. Linker uses this to update the microcode image.
- ‘C’ Compiler variables memory segments locations and sizes.
- Message library/function dispatches. For service thread routines, a function call in an application will result in a jump to a microPC location for a Microengine-thread. The microcode page and label must be bound with the application library and function pointer.

### 5.1.2 Shared Address Update (Flow)

1. From Assembler, get a list of externs. Get the microPC locations where these variables are used as an immediate. Get the microword format to know offset and size of the immediate. Format this in the microcode “linked” object.
2. In the StrongARM Application, load or map the microcode “linked” object.
3. In the StrongARM application, bind the external variable with a value by calling the appropriate function in the loader library.
4. The loader library updates the “immediates” with the bind-value for all occurrences of the variable in the micro-image.
5. In StrongARM application, load the images to appropriate Microengines via a loader library function.

## 5.2 Microengine Image Linker (UCLD)

ucld is an executable that accepts a list of Microengine images (*.list) generated by the Assembler, uca, and combines them into a single object that is loadable by the core image, running on the StrongARM processor, utilizing Microengine Loader Library (libD) functions.

Usage: `ucld [options ...] uca_list_file ...`

Command line options:

`-h`
Print a description of the ucld commands.

`-o outfile`
Produce an output object file of the name outfile. The default output name is ucld.uof.

`-u Microengines`
Associate the specified Microengine(s) to subsequent uca_list_file. The Microengines option is a string of digits 0 through 5, with no space separating the digits, representing the number of the Microengine. The -u option can be repeated for subsequent list files. The default is '012345' which associates all Microengines to the uca_list_file that follows.

`-f [fill_pattern]`
Output all 1024 microwords and fill the unused microwords with the four-byte hexadecimal constant specified by fill_pattern. If the -f option is not specified, then only up to the last microword used will be output. The default fill_pattern value is 0x0.

`-c`
Creates a hexidecimal representation of the output object to a ‘C’ module. The filename will be the name of the output object file but with a ‘.c’ file type.

`-i`
Ignore CPU versions compatibility error.

`-g`
Include debugging information, to be used by the Workbench, in the output object file.

`-v`
Print a message that provides information about the version of the Linker being used.

## 5.3 Generating a Microengine Application

On development system:

```
% uca ueng0.uc -o ueng0.list
% uca ueng1_5.uc -o ueng1_5.list
% ucld -u 0 ueng0.list -u 12345 ueng1_5.list -o ueng.uof
```

On an ARM machine:

```
arm> armcc -o core_app core_app.c uclo.a
arm> core_app ueng.uof
```

## 5.4 Syntax Definitions

### 5.4.1 Image Name Definition

This definition associates a name to the content of the uca_list_file in the output object file. This provides the means of identifying the image within the object file, allowing referencing to particular section of the object file by name - used by the loader link library. The image name must be unique for all input files that are to be linked. The Image name is defined in the micro-code source file (*.uc) use of the .image_name keyword. If the .image_name is not specified in the micro-code source file, then the image name will be the name of the uca_list_file excluding any folder path and the file extension.

Format for uca source (*.uc), and output (*.list) files:

```
.image_name name
```

### 5.4.2 Import Variable Definition

This scheme provisions the sharing of address pointers between Microengine images and the core image.

An IXP1200 application is physically comprised of two parts: a Microengine image and a core image. Microengine images are microinstructions that run on the Microengine and are created using the uca and ucld tools. Core image runs on the StrongARM Core processor and is created using the ARM compiler/assembler and linker tools.

Address sharing between the Microengine and core images is achieved by declaring the variables as an imported/external variable using the .import_var keyword in the microcode source file (*.uc), prior to the variables being used. The uca assembler will create a list of microword addresses and the field bit positions within the microword where the variables are used and provided the information in its output file (*.list) in the format as described below. The ucld Linker will process the uca output files (*.list), possible one for each Microengine, and store the information in a format that is acceptable by the loader. Core application binds and resolves a value to the Import-Variable, by using the functions provided in the Micro-code Loader Library (uclo.a). The binding/ resolution of import variables must be performed prior to the loading of the micro image to the Microengines.

Format for uca source (*.uc) file:

```
.import_var variable_name variable_name …
```

Format of uca output (*.list) files:

```
.%import_var variable_name page_id uword_address
<msb:lsb:rtSht_val, ...msb:lsb:rtShf_val>
```

Where:

**`variable_name`** String name of the external variable as declared in the uca source file (*.uc).

**`highLow_flag`** An integer 0 or 1 indicating the lower or upper respective 16 bits of the external 32-bit address.

**`page_id`** String name of the page identifier

**`uword_address`** An integer number between 0 and 1023 indicating the micro word address.

**`<msb:lsb:rtSht_val,...msb:lsb:rtShf_val>`**
A list of a maximum of four integer triad representing the bit field most and least significant bit positions, and the number of bit position to right shift the source address value.

## 5.4.3 Microengine Assignment

This feature allows the loading of Microengine images to the appropriate Microengine at load time. This is achieved by the specifying the -u option on the command line indicating the number of Microengine(s) to be associated with the subsequent uca-list-file. The -u option can be repeated for subsequent input files, and defaults to '012345' if the option is not specified. An error will be generated if a Microengine is assigned to multiple input files.

Format of command line to associate Microengines 0, 1, and 2 to the input file test.list, and to associate Microengines 3, 4, and 5 to the input file test2.list:

```
ucld -u 012 test.list -u 345 test2.list
```

## 5.4.4 Thread Type Definition

Thread type definition allows the user to specify whether the Microengine thread is either an autonomous or a service type. An autonomous type thread is one that typically has one entry point and implements a specific task/operation. A service type thread image typically contains multiple entry points -- similar to dynamic library. Thread_ids is a string of decimal digits, 0 through 3 without spaces separating the digits, identifying the thread context. Undefined thread type defaults to an Autonomous type.

Format for source (*.uc), and output list (*.list) files:

```
.thread_type thread_ids ["SERVICE" or "AUTONOMOUS"]
```

## 5.4.5 Code Entry Point Definition

Code entry point definition allows the user to specify the starting point of each Microengine image instead of the default entry point of zero. This definition is of particular use to autonomous type thread--service type threads have multiple entry points that are defined by ".export_func" keyword.

Format for uca source file (*.uc), and output (*.list) files:

```
.entry page_id uword_address
```

### 5.4.6 Export Function Definition

The exporting of Microengine functions, allows the invocation of these functions from the core image. Export functions are addressed relative to a function jump-table. The association between function jump table and functions is achieved by prefixing the table name to the function name. Function jump tables are defined by the .func_table keyword followed a label. Export functions are defined using the .export_func keyword preceded by a definition of a function jump-table. Export functions must be within 256 microwords of the their related table-maximum of 256 export functions per function table. Therefore, care must be taken to avoid the placing of extraneous microinstructions between the table definition and export functions definition that will reduce the total number of functions in the table.

Func_type is either the keyword one way or two way, which specifies whether or not the function returns a value.

Thread_ids is a string of decimal digits, between 0 and 3 without spaces separating the digits, identifying the Microengine thread(s) that could be used to execute the defined function(s).

In_arg_len and out_arg_len specify the number long-words that comprised the input and output arguments respectively. Argument lengths should be have a value between 0 and 3 long-words.

Format for uca source file (*.uc):

```
.func_table tableName
.export_func func_type thread_ids tableName_funcName in_arg_len out_arg_len
:
.export_func func_type thread_ids tableName_funcName in_arg_len out_arg_len
```

Format for uca output list file (*.list):

```
.%func_table tableName page_id uword_addr
.%export_func func_type thread_ids tableName_ funcName in_arg_len out_arg_len
page_id uword_addr
:
.%export_funcl func_type thread_ids tableName_ funcName in_arg_len out_arg_len
page_id uword_addr
```

## 5.5 Examples

### 5.5.1 Uca Source File (*.uc) Example

```
.entry common_code 0
.thread_type 0123 service
.image_name test
.import_var rbuf rswap
.func_table memsv#
.export_func oneway 0123 memsv_bcopyt# 1 1
.export_func oneway 0123 memsv_bzero# 2 0
.memsv#:
memsv_bcopyt#:
br[memsv_bcopysubr#]
:
memsv_bzero#:
br[memsv_bzerosubr#]
```

```
memsv_bcopy#:
br[memsv_bcopysubr#]
```

### 5.5.2 Uca Output File (*.list) Example

```
.entry common_code 0
.thread_type 0123 service
.image_name test
 .%import_var rbuf common_code 1 <16:0:0>
.%import_var rbuf common_code 2 <31:16:16> 10<31:16:16> 21<31:16:16>
.%import_var rswap common_code 3 <16:0:0>
.%import_var rswap common_code 4 <31:16:16>
.%export_func oneway 0123 memsv_bcopyt# 1 1 common_code 44
.%export_func oneway 0123 memsv_bzerot# 1 1 common_code 45
:
memsv#:
memsv_bcopyt#:
br[memsv_bcopysubr#]
.44 F8001C07 common_code
memsv_bzero#:
br[memsv_bzerosubr#]
.45 F8002707 common_code
memsv_bcopy#:
br[memsv_bcopysubr#]
.46 F8003E87 common_code
:
```

## 5.6 Microcode Object File (UOF) Format

**Table 5-1. FileHdr—File Header**

| | |
|---|---|
| FileID | 8 bytes: value identifying the file as a chunk formatted file. |
| MaxChunks | 2 bytes: maximum possible chunks that the file can contain-specify at the creation of the file. |
| NumChunks | 2 bytes: number of chunks currently being used. |
| File-chunk headers | MaxChunks * sizeof(FileChunkHdr) contiguous bytes of file chunk headers. |

Identifies the file format. This header is mandatory and must be the first entry in the file and consists of a fixed 16-byte section and a variable section indexing the chunks.

**Table 5-2. FileChunkHdr**

| | |
|---|---|
| FileChunkId | 8 bytes: a unique value identifying the chunk |
| Checksum | 4 bytes: CRC checksum of chunk. |
| Offset | 4 byte: offset into the file where the chunk begins |
| Size | 4 bytes: size of the chunk. |

Variable section of the File Header.

### 5.6.1 FileChunkId Type: UOF_OBJS

**Table 5-3. Objects Header - UOF_OBJS - UOF**

| | |
|---|---|
| CpuType | 4 bytes: CPU family type. |
| CpuMinVers | 2 bytes: the minimum CPU revision that the UOF will run on. |
| CpuMaxVers | 2 bytes: maximum CPU revision that the UOF will run on. |
| MaxChunks | 2 bytes: maximum chunks that can be contained in an UOF chunk. |
| NumChunks | 2 bytes: number of chunks currently being used. |
| Reserved1 | 4 bytes: reserved for future use. |
| Reserved2 | 4 bytes: reserved for future use. |
| UOF-object headers | MaxChunks * sizeof(UofChunkHdr) contiguous bytes of uof object headers. |

**Table 5-4. Ucode Object Header - UofChunkHdr**

| | |
|---|---|
| ObjId | 8 bytes: a unique value identifying the object |
| Offset | 4 byte: offset of the object from the beginning of the UOF object |
| Size | 4 bytes: size of the object |

### 5.6.2 UOF ObjId Types: UOF_STRT, UOF_GTID, and UOF_IMAG

**Table 5-5. String Table Containing the Ucode Object Strings - UOF_STRT**

| | |
|---|---|
| TableLength | 4 bytes: total length of the strings |
| Strings | NULL terminated strings |

**Table 5-6. Generating Tool Identification - UOF_GTID**

| | |
|---|---|
| ToolName | 8 bytes: generating-tool name offset into the UOF string table |
| ToolVersion | 4 bytes: generating-tool version |
| Reserved1 | 4 bytes: reserved for future use. |
| Reserved2 | 4 bytes: reserved for future use. |

**Table 5-7. Image Module - UOF_IMAG**

| Image Module | |
|---|---|
| ImageName | 4 bytes: image name offset into UOF string table |
| FillPattern | 4 bytes: unused ustore fill pattern |
| Reserved1 | 4 bytes: reserved for future use. |
| Reserved2 | 4 bytes: reserved for future use. |
| CpuType | 4 bytes: CPU family type. |
| CpuMinVers | 2 bytes: minimum CPU revision that the image will run on. |

### Table 5-7. Image Module - UOF_IMAG

| | |
|---|---|
| CpuMaxVers | 2 bytes: maximum CPU revision that the image will run on. |
| ImageAttrib | 2 bytes: uEngineFlags(byte 0), threadType(byte 1<3:0>, unused(byte 1<7:4>) |
| EntryPage | 2 bytes: page number entry point |
| EntryAddress | 2 bytes: uPC entry point into the code |
| NumOfPages | 2 bytes: the number of page |
| Micro store pages | NumOfPages * sizeof(CodePage) contiguous bytes of code pages |

### Table 5-8. Microword Page - CodePage

| | |
|---|---|
| ImpVarTabSize | 2 bytes: size of the import variable table |
| CodeAreaSize | 2 bytes: size of the code area |
| ExpFuncTabSizee | 2 bytes: export func table size |
| Unused1 | 2 bytes: reserved/alignment |
| Reserved1 | 4 bytes: reserved for future use. |
| ImpVarTableOffset | 4 bytes: offset into the file of the import variable table |
| CodeAreaOffset | 4 bytes: offset into the file of the code area |
| ExpFuncTabOffset | 4 bytes: offset into the file of the export function table |

### Table 5-9. Import Variables Table - ImpVarTab

| | |
|---|---|
| NumEntries | 4 bytes: the number of entries in the table |
| Table entries | NumEntries * sizeof(ImportVar) contiguous bytes of import variables |

### Table 5-10. Import Variables - ImportVar

| | |
|---|---|
| VarName | 4-bytes: offset into string table |
| AddressDirective | 4-bytes: micro word address(bytes 0 &1), unused (bytes 2 & 3) |
| FieldPos | 12-bytes: fields position within uword; each byte represents a bit position, field length, and a right-shift value - max of four fields |

### Table 5-11. Microwords - CodeArea

| | |
|---|---|
| NumMicroWords | 4 bytes: the number of uwords in segment 1 |
| NumMicroWords_2 | 4 bytes: the number of uwords in segment 2. |
| Micro words | (NumMicroWords + NumMicroWords_2) * 4 contiguous bytes of micro words |

### Table 5-12. Export Function Table - ExpFuncTab

| | |
|---|---|
| NumEntries | 4 bytes: the number of entries in the table |
| Table entries | NumEntries * sizeof(ExportFunc) contiguous bytes of export function labels |

**Table 5-13. Export Functions - ExportFunc**

| | |
|---|---|
| FuncName | 4 bytes: function name offset into UOF string table |
| FuncAddress | 2 bytes: micro word address-relative to the function table |
| ThreadIds | 2 bytes: threadID flags(byte 0<3:0>), funcType(byte 1<1:0>), unused (bytes 0<7:4>, 1<7:2>) |
| InArgLen | 2 bytes: input arguments length in long-words |
| OutArgLen | 2 bytes: output arguments length in long-words |

## 5.6.3 FileChunkId Type: DBG_OBJS

**Table 5-14. Debug Objects Header - DBG_OBJS**

| | |
|---|---|
| MaxChunks | 2 bytes: maximum objects that can be contained in a DBG_OBJ chunk. |
| NumChunks | 2 bytes: number of chunks currently being used. |
| Debug-object headers | MaxChunks * sizeof(DbgChunkHdr) contiguous bytes of debug object headers. |

**Table 5-15. Debug Object Chunk Header - DbgChunkHdr**

| | |
|---|---|
| ObjId | 8 bytes: a unique value identifying the object |
| Offset | 4 byte: offset of the object from the beginning of the Debug object |
| Size | 4 bytes: size of the object |

## 5.6.4 DBG ChunkId Types: DBG_STRT, DBG_IMAG

**Table 5-16. Debug Object Strings - DBG_STRT**

| | |
|---|---|
| TableLength | 4 bytes: total length of the strings |
| Strings | NULL terminated strings |

String table containing the Debug Object Strings.

**Table 5-17. Debug Image - DBG_IMAG**

| | |
|---|---|
| ImageName | 4 bytes: list-file name string-table offset |
| ImageAttrib | 2 bytes: uEngineFlags(byte 0), unused(byte 1) |
| DbgRegTabSize | 2 bytes: size of the Register table |
| DbgLblTabSize | 2 bytes: size of the Label table |
| DbgSrcTabSize | 2 bytes: size of the Source table |
| DbgRegTabOffset | 4 bytes: register table offset from beginning of the debug object |
| DbgLblTabOffset | 4 bytes: label table offset from the beginning of the debug object |
| DbgSrcTabOffset | 4 bytes: source table offset from the beginning of the debug object |

Contains debug information from a particular list file.

### Table 5-18. Debug Object Table - DbgObjTable

| | |
|---|---|
| NumEntries | 2 bytes: the number of objects in the table |
| Unused1 | 2 bytes: reserved/alignment |
| Table entries | NumEntries * sizeof(object) contiguous bytes of objects |

Table containing either Registers, Labels, or Source objects.

### Table 5-19. Debug Registers - DbgReg

| | |
|---|---|
| Name | 4 bytes: offset into string table |
| Type | 2 bytes: register type |
| Addr | 2 bytes: address of the register |

### Table 5-20. Debug Labels - DbgLabels

| | |
|---|---|
| Name | 4 bytes: label name offset into the debug string table |
| Addr | 2 bytes: address of the label |
| Unused1 | 2 bytes: reserved/alignment |

### Table 5-21. Debug Source Lines - DbgSource

| | |
|---|---|
| FileName | 4 bytes: filename offset into debug string table |
| Lines | 4 bytes: source lines offset into the string table |
| LineNum | 2 bytes: line number |
| Addr | 2 bytes: the associated microcode address |
| ValidBkPt | 2 bytes: valid breakpoint indicator |
| Unused1 | 2 bytes: reserved/alignment |

# Transactor 6

This chapter describes the IXP1200 Transactor and its command line interface. The IXP1200 Workbench graphical user interface to the Transactor is described in Chapter 2. The chapter contains the following sections:

- Overview (see Section 6.1).
- Invoking the Transactor (see Section 6.2).
- Command Interface (see Section 6.3).
- IXP1200 Transactor Commands (see Section 6.4).
- C Interpreter (see Section 6.5).
- Debugging (see Section 6.6).
- Transactor Command Script Files (see Section 6.7).
- Enabling the StrongARM Core (see Section 6.8).
- Simulation Switches (see Section 6.9).

## 6.1 Overview

The IXP1200 Transactor executes IXP1200 Microengine object code. It demonstrates the functional behavior and performance characteristics of a system design based on the IXP1200 without relying on IXP1200 hardware. The IXP1200 Transactor is a cycle-accurate architectural model of the IXP1200 hardware that is optimized for high-speed simulation of an IXP1200 based system.

The first step in using the IXP1200 Transactor is to create a model of an IXP1200 based system. This involves the use of commands that define how many IXP1200 chips are in the system and the amount of SRAM and SDRAM associated with each IXP1200. Once the system model has been defined, the Transactor can be used to run simulations, debug them, and gather performance statistics.

The Transactor recognizes two types of input:

- **Transactor commands.** Commands unique to the IXP1200 Transactor that allow you to control its operation.
- **A subset of C commands.** Provided to give you greater flexibility in controlling the Transactor.

Commands can be typed in through the command line interface or executed as a series of commands from a command script file.

## 6.2 Invoking the Transactor

The IXP1200 Transactor is invoked by using a command in the following format:

```
safe_xact {switch1 switch2 switch3}
```

Up to three optional switches can be specified in any order. These switches take the following forms:

**Table 6-1. IXP1200 Transactor Optional Switches**

| | |
|---|---|
| `-i command_script.ind` | Execute the command script file command_script.ind |
| `-m directory_path uword.def` | Look for the file uword.def in the folder specified by directory_path. This file is required to invoke the IXP1200 Assembler with the uca Transactor command when the Transactor is running. If the file is located in the same folder as safe_xact.exe, this switch is not required to run the Assembler |
| `-p directory_path ucc.def` | Look for the file ucc.def in the folder specified by directory_path. This file is required to invoke the IXP1200 Assembler with the uca Transactor command when the Transactor is running. If the file is located in the same folder as safe_xact.exe, this switch is not required to run the Assembler. |

## 6.3 Command Interface

The command interface performs the following functions:

- Enable state initialization and model configuration.
- Model simulation.
- State assignment or examination.
- Statistics collection and reporting.
- Microcode debug.
- Command file execution
- Data logging.
- Model save/restore capability.

Commands are interpreted as either a C statement or expression (when terminated with a semicolon or a "}") or as an transactor-specific command otherwise.

A subset of the C preprocessor function allows users to define macros using the "#define" command, thereby allowing users to customize the command interface.

The C interpreter implements a subset of the ANSI C language definition. The following summarizes its capabilities:

- Handles "if", "if/else", "while", "for", "break", "continue", "return", code blocks (i.e. list of C statements delimited by "{}"), structs, and most C expressions.
- Any simple model state (i.e. a signal, register or array (not a fifo) is treated as a global int when it appears in a C statement or expression. Users can additionally create user-defined int variables (currently, unsigned int, char, etc., are not supported).
- As in C, scoping of variables is implemented based on their definitions within {...}. Variables defined outside of any {...} are defined as global variables and can also be examined and deposited by non-C commands.

- Functions may be defined and called by the user in the normal manner. Some built-in C functions are also provided (see "help built-in C functions" for more information). User-defined C functions with variable argument lists are currently not supported.
- C "struct" definitions can be defined and instantiated in both SRAM and SDRAM memory spaces.

Note that when C text is interpreted from the command line, it assumes the entire C expression is specified when the user presses the ENTER key. The user can specify code on multiple lines by entering a continuation character that instructs the interpreter to wait for more input before interpreting the text. Continuation characters come in two flavors:

1. A "\" at the end of a line instructs the interpreter to wait for the subsequent line to be input,
2. An implicit continuation always exists as long as one or more "{" are not terminated with "}".

## 6.4 IXP1200 Transactor Commands

Transactor commands fall into four functional categories: initialization, simulation, debugging, and miscellaneous. The sections that follow list the Transactor commands in alphabetical order. Each command description includes:

- The symbolic command name (e.g., **examine**).
- A short description of what the command does.
- The format of the command.
- Command parameter descriptions, and where appropriate, definitions of predefined parameters.
- One or more examples illustrating the use of the command.
- Optional input strings delimited by bracket characters ({}).
- The table that follows is a quick reference to the IXP1200 Transactor commands:

### Table 6-2. IXP1200 Transactor Commands (Sheet 1 of 2)

| Command Name | Command Type | Section |
|---|---|---|
| #define | Debugging | 6.4.8 |
| #elifdef | Debugging | 6.4.10 |
| #else | Debugging | 6.4.11 |
| #endif | Debugging | 6.4.12 |
| #ifdef | Debugging | 6.4.20 |
| #ifndef | Debugging | 6.4.21 |
| #undef | Debugging | 6.4.40 |
| @ | Miscellaneous | 6.4.1 |
| bank_analysis | Initialization | 6.4.2 |
| chip | Initialization | 6.4.3 |
| close | Debugging | 6.4.4 |
| config | Miscellaneous | 6.4.5 |
| connect | Miscellaneous | 6.4.6 |
| debug | Debugging | 6.4.7 |
| deposit | Debugging | 6.4.9 |
| examine | Debugging | 6.4.11 |
| exit | Miscellaneous | 6.4.14 |
| fbox | Debugging | 6.4.15 |
| go_clk_domain | | 6.4.16 |
| go | Simulation | 6.4.17 |
| goto | Simulation | 6.4.18 |
| help | Miscellaneous | 6.4.19 |
| init | Initialization | 6.4.20 |
| load_bin_file | Miscellaneous | 6.4.23 |
| load_list_file | Miscellaneous | 6.4.24 |
| load_uc | Initialization | 6.4.25 |
| load_uof_file | Initialization | 6.4.26 |
| log | Debugging | 6.4.27 |
| mem_init | Initialization | 6.4.28 |
| path | Debugging | 6.4.29 |
| restore | Debugging | 6.4.30 |
| return | Miscellaneous | 6.4.31 |
| save | Debugging | 6.4.32 |
| set_clk_freq | Initialization | 6.4.33 |
| sim_reset | Simulation | 6.4.34 |
| statistics | Debugging | 6.4.35 |
| time | Miscellaneous | 6.4.36 |

**Table 6-2. IXP1200 Transactor Commands (Sheet 2 of 2)**

| Command Name | Command Type | Section |
|---|---|---|
| trace | Debugging | 6.4.37 |
| ubreak | Debugging | 6.4.38 |
| uca | Miscellaneous | 6.4.39 |
| version | Miscellaneous | 6.4.41 |
| watch | Debugging | 6.4.42 |

## 6.4.1 @

Executes a series of simulation commands in a specified command script file name.

**Format:** @cmd_file_name

**`cmd_file_name`**

Name of command script file. See Section 6.7 for more information on Transactor command script files.

**Example:**
```
> @myfile.ind
```
Begins execution of the command script file named myfile.ind.

## 6.4.2 bank_analysis

When executed the first time, enables SDRAM reference statistics gathering to be used so that SDRAM bandwidth can be improved by setting the position of the SDRAM bank bits.

At the end of the simulation, the command can be executed again to display the banking analysis results.

**Format:** `bank_analysis`

**Example:**
```
> bank_analysis
```
Begins bank analysis or display of statistics results.

The overall goal of the SDRAM controller design is to maximize memory bandwidth. Since multiple D-streams are simultaneously processed by the SDRAM controller, the likelihood of subsequent references mapping to the current open row of the current bank is very low. Therefore, rows are scheduled for precharging when a reference completes so that row will be ready for another RAS as soon as possible (NOTE: the one exception to this rule is when references are explicitly “chained” together. In this case the row is held open for the reference following the chained reference). Thus, the preferred access pattern employed by the SDRAM controller is to process references in an order which accesses a different SDRAM bank every reference. By doing this, the precharge time of one bank can be hidden under the execution of another reference. The more often this pattern occurs, the greater the realized memory bandwidth. In order to maximize this access pattern the SDRAM controller can swap the positions of one or two bank bits with one or two other specified bits. By transposing the proper bits, the probability of bank swapping between adjacent references can be increased, thus increasing overall memory bandwidth.

At the end of a simulation, the bank_analysis command is used to gather SDRAM reference statistics to determine which bits should be swapped with the actual bank bits in order to optimize memory bandwidth for the particular application running.

### 6.4.3 chip

Adds an instance of the IXP1200 chip to the simulation environment. Each instantiated chip is automatically connected to a common FBUS and a common PCI bus, but each chip has its own SRAM and SDRAM memories.

Each instantiated chip must have a unique name comprised of alphanumeric characters or the underscore ( _ ) character. The name cannot begin with a numeric character nor can its name be the same as one of the hierarchical instance name of the chip's immediate children (e.g., sc for SRAM controller). A null name can be specified by leaving **chip_name** blank.

**Format:** `chip {/rdybus=chip_name} {/fbus=chip_name} {chip_name}`

**`chip_name`** An optional chip name. If no name is specified, the chip defaults to a null name.

**`/rdybus`** Allows the user to configure the connectivity of one or more Ready Buses among multiply instantiated IXP1200 devices. If this switch is not specified, the instantiated IXP1200 Ready Bus is connected to the globally shared Ready Bus called rdybus (with no chip prefix). If this switch is never used when instantiating multiple chips, all chips are connected to the same Ready Bus. If a chip command is specified as chip/rdybus=A B for example, then the Ready Bus of chip B becomes the same as the Ready Bus of chip A (i.e., A.rdybus). In that case, chip A must be defined prior to executing this command.

**`/fbus`** This switch works analogously to /rdybus except that it uniformly applies to the following buses composing the FBus interface: fdat_hi, fdat_lo, fbe_hi_l, fbe_lo_l, sop, eop, token, fps, portctl_l, txasis.

**Example 1:**
```
> chip a0
```
Instantiates a chip named a0.

**Example 2:**
```
> chip
```
Instantiate a chip with a null name.

### 6.4.4 close

Closes a previously opened log or trace file. Exiting the Transactor will automatically close all open log or trace files.

**Format:** `close filename`

**`filename`** `Name of the log or trace file.`

**Example:**
```
> close mylog.txt
```
Close a log file.

### 6.4.5 config

Displays the current simulation configuration.

**Format:** `config`

**Example:**
```
> config
```

**Output:**

```
Current Simulation Configuration:  1 SA-1200 chip
============================================================
chip:  ""
sram:  memory array of 524288 32-bit words (2 MB);  545 words
currently valid.
flash:  memory array of 524288 32-bit words (2 MB);  0 words
currently valid.
SRAM priority queue depth:    8
SRAM read queue depth:        16
SRAM order queue depth:       16
SRAM read lock queue depth:   24
```

## 6.4.6 connect

Effectively “wires” together the specified FBus input and output pins so that the input pin state is assigned the value that the output pin state received during the previous FBus cycle.

**Format:** `connect fbi_input_pin fbi_output_pin`

**`fbi_input_pin`**
Name of FBus input pin.

**`fbi_output_pin`**
Name of FBus output pin.

**Example:**

```
> connect pin1 pin2
```

## 6.4.7 debug

Displays debugging information related to the Microengines, SDRAM, SRAM, and the FBI. The information displayed varies according to the settings specified by the unit.art.debug and fx.art.debug_px states (see Appendix A for definitions of these states).

**Format:** `deb{ug} {/status}`

**`/status`** Displays the current state of all debug enable switches.

**Example 1:**

```
> debug /s
f0.art.debug:     enabled  (FBOX info)
f0.art.debug_p0:  enabled  (FBOX pipe 0 info)
f0.art.debug_p1:  enabled  (FBOX pipe 1 info)
f0.art.debug_p2:  enabled  (FBOX pipe 2 info)
f0.art.debug_p3:  enabled  (FBOX pipe 3 info)
f0.art.debug_p4:  enabled  (FBOX pipe 4 info)
f1 is not currently enabled for simulation
f2 is not currently enabled for simulation
f3 is not currently enabled for simulation
f4 is not currently enabled for simulation
f5 is not currently enabled for simulation
sc.art.debug:  disabled  (SRAM controller info)
dc.art.debug:  disabled  (SDRAM controller info)
```

**Example 2:** Perform debugging with state settings made to display all five execution stages of a Microengine.

```
> debug
====P4 fbox stage (write result) for f0. in ctx: 0 at fbox cycle 5====
uword addr = 0
.operand_synonym  no_of_links 0x3
start#:
    ld_field_b[cur_ptr, 1111, headptr, 0] ;cur_head = headptr
W_BUS:  00000001
operand write:  GPR_A[0]=00000001
====P3 fbox stage (ALU/shifter) for f0. in ctx: 0 at fbox cycle 5====
uword addr = 1
    immed[loop_no, no_of_links]     ;loop = the number of links
A operand: 00000000; pre-shift B operand: 00000003; B operand:
00000003
shifter op:  left shift 0;  ALU op:  ALU_OP_B;
ALU result: 00000003; old_ccs:  !=0:>=0:no_carry
====P2 fbox stage (operand lookup) for f0. in ctx: 0 at fbox cycle 5===
uword addr = 2
loop_add#:
    alu_shf_ri[val_ptr,cur_ptr, +, 1, 0]  ;val_ptr = cur_ptr +1
A operand:  00000001  (selected from writeback bypassed data);
B operand:     00000001  (selected from immed data);
====P1 fbox stage (initial decode) for f0. in ctx: 0 at fbox cycle 5===
uword addr = 3
    sram_bi[read, $next, cur_ptr, 0, word_cnt_1]  ;$next = next
====P0 fbox stage (ustore lookup) for f0. in ctx: 0 at fbox cycle 5====
ustore addr = 4 ==> A00A3038
    sram_bi[read, $val, val_ptr, 0, word_cnt_1],
defer[1], ctx_swap      ;$val = valdep art.ctx_mask = 0x8
```

## 6.4.8 #define

Implements the text substitution function of the C preprocessor. Allows a user to specify text that is automatically substituted into every command line before the line is interpreted. This allows users to customize the command interface.

When a macro has substitutable arguments three preprocessor operators may be applied to modify the resulting string. The preprocessor operator **##** can be inserted between two tokens in the macro_text to cause the two adjacent tokens to be concatenated. The preprocessor operator **#** can be prepended onto a macro_text token causing the following token to be quoted. The preprocessor operator **###** can be prepended causing the following token to be dequoted if quote characters exist as the first and last characters of the token.

Whenever a preprocessor operator is applied, the one or two tokens associated with the operator are not expanded. In all other cases, every token in the macro_text is recursively expanded (if possible) based on other existing macro definitions. If the recursive expansion of a macro name yields a token of the same name, no further expansion occurs.

**Format:**

```
#define macro_name {macro_argument_1,
macro_argument_2, macro_argument_3,...}
macro_text
```

**`macro_name`** Name of macro to be defined. If the macro_name has arguments associated with it, the arguments are substituted into the macro_text

whenever the corresponding formal argument name matches a token of the macro_text.

**Example:**

```
> #define g go
```

## 6.4.9 deposit

Evaluates the C numeric expression deposit_expr and deposits the resulting number into the specified state. The wildcard asterisk character (*) can be used to avoid specifying the entire state name. However, unless the /multiple qualifier is specified, the wildcard specification must unambiguously address only one state value.

The C interpreter only supports 32-bit integers during expression and statement evaluation. Use the **deposit** command to initialize 64-bit data elements by depositing the data 32 bits at a time (see example below).

**Format:** dep{osit} {deposit_qualifiers}
state_spec{[index_range]}{<bit_range>} = deposit_expr

**deposit_qualifiers**
/silentInhibits reporting the deposit action.
/multipleAllows a wildcarded name to deposit to multiple states.

**state_spec** Any predefined simulation state or user defined state that holds a numeric value. See Appendix A for more details on states.

**index_range** The index_range specification is only relevant for arrays or FIFO structures and can be a single C numeric expression or two numeric expressions separated by a semicolon (:) to indicate the inclusive range formed between the two numbers. For FIFOs, [ [index_range] causes the structure to be addressed in a first-in-first-out manner, where 0 represents the first element, 1 the second, and so on. The index range can also be specified in terms of [beginning:+length], where the + following the colon indicates that the end of the index range is determined by adding the specified length to the beginning.

**bit_range** The bit_range spec has the same form as the index_range. If it is not specified, the whole field is assumed.

**deposit_expr** A C numeric expression.

**Examples:**

```
> dep sram [1:20]=1

> dep sdram [1:+10] = 0xabcd

> dep f0.reg.reg_name_0 = 5

> dep f1.reg.@absolute_reg_name = 6

> dep f0.art.thread_start_addr =
uaddr(f0.ctl.ustore,"label1#")

> dep/m f0.art.debug*1

> sdram[50010] = (0x55667788 << 16);
> dep/s sdram[50011]<47:16> = 0x55667788
> ex sdram [50010:50011]
```

```
> memory:sdram[50010]<63:0>: 00000000 77880000
> memory:sdram[50011]<63:0>: 00005566 77880000
```

## 6.4.10 #elifdef

The #elifdef (else ifdef) directive, with the #ifdef, #ifndef, #else and #endif directives, controls execution of commands typed at the command line or in a script file. If the symbol following #elifdef is defined AND the preceding #ifdef/#ifndef and #elifdef's are not satisfied, the commands immediately following the #elifdef directive are executed up to the next #else, #elifdef, or #endif directive.

An #elifdef directive must be preceded by an #ifdef or #ifndef directive. Any number of #elifdef directives can appear between the #ifdef and #endif directives.

**Format:** `#elifdef macro_name`

## 6.4.11 #else

The #else directive, with the #ifdef, #ifndef, #elifdef and #endif directives, controls execution of commands typed at the command line or in a script file. If the symbols in the closest preceding #ifdef and #elifdef's are not defined, the commands immediately following the #else directive are executed up to the next #endif directive.

Any number of #elifdef directives can appear between the #ifdef/#ifndef and #endif directives, but at most one #else directive is allowed. The #else directive, if present, must be the last directive before #endif.

## 6.4.12 #endif

The #endif directive, with the #ifdef, #ifndef, #elifdef and #else directives, controls execution of commands typed at the command line or in a script file. Each #ifdef/#ifndef directive must be matched by a closing #endif directive.

## 6.4.13 examine

Examines the current state of one or more simulation states or user-defined variables. The wildcard asterisk character (*) can be used to specify multiple states to be examined.

**Format:**
```
ex{amine} {qualifier_list}
state_spec{[index_range]}{<bit_range>}
```

**`qualifier_list`**

Optional qualifiers may be applied to constrain the examination of multiple variables to specific state types defined by the qualifiers. Any number of qualifiers may be applied. They are:

/arrayData array.
/artifactModel artifact state (does not model real hardware).
/delayBus transfer delay element.
/fifoQueue structure.
/functionUser-defined C function.
/registerFlip-flop or latch hardware state element of 32 bits or less.

/signalCombinatorial hardware state element of 32 bits or less.
/statisticPerformance data collection and display facility.
/structUser-defined C struct definition.
/ustoreMicrocode control store.
/variableUser-defined C variable.
/watchUser-defined watch function.

**state_spec** Any predefined simulation state or user defined state that holds a numeric value. See Appendix A for more details on states.

**index_range** The index_range specification is only relevant for arrays or FIFO structures and can be a single C numeric expression or two numeric expressions separated by a semicolon (:) to indicate the inclusive range formed between the two numbers. For FIFOs, [ [index_range] causes the structure to be addressed in a first-in-first-out manner, where 0 represents the first element, 1 the second, and so on. The index range can also be specified in terms of [beginning:+length], where the + following the colon indicates that the end of the index range is determined by adding the specified length to the beginning.

**bit_range** The bit_range spec has the same form as the index_range. If it is not specified, the whole field is assumed.

**Example:**

```
> examine f*abort
> examine/stat *
> examine/stat f0.*
> ex sdram [0:+20]
```

## 6.4.14 exit

Closes all open log files and then exits the Transactor.

**Format:** `exit`

**Example:**

```
> exit
```

Exit the Transactor

## 6.4.15 fbox

Displays a summary of Microengine execution status.

**Format:** `f{box}`

**Example:**

```
>fbox
Fbox:  f0
ctx_0:  saved_pc:  5;  signals--> SELF ;  wake_enable-->
ctx_1:  saved_pc:  0;  signals--> SELF ;  wake_enable-->
ctx_2:  saved_pc:  0;  signals--> SELF ;  wake_enable-->
ctx_3:  saved_pc:  0;  signals--> SELF ;  wake_enable-->

ping ref (output):  empty
pong ref (input ):  empty
------------------------------------------------------------------------
0 PCIT master ops pending completion.
Simulation Time:  87 (core cycles); 43 (f_bus cycles) 43 (pci cycles)
```

```
NOTE:  "double click" CTRL-C to halt simulation.
```

Simulation time cycles for Core, FBus, and PCI are gathered from the following model variables:

```
sim.core_clk_cycle<31:0>
sim.fbus_clk_cycle<31:0>
sim.pci_clk_cycle<31:0>
```

## 6.4.16 go_clk_domain

Sets the clock domain default used in the "go" command when simulating a cycle count. Valid values for the clock domain are: "fbox", "fbus", or "pci".

**Format:** `go_clk_domain clock_domain_name`

## 6.4.17 go

Simulates the number of Microengine cycles or microinstructions specified by **count**. If no count is given, count defaults to 1.

**Format:** go{/silent} {fbox_name/thread_num} {count}

**`/silent`** Suppresses debug information during the simulation.

**`fbox_name/thread_num`**
The simulator runs until the specified number of microinstructions has executed for the specified Microengine/thread. In other words, this qualifier allows one to step through the execution of a particular thread in the presence of multiple thread context switches.

**`count`** Number of Microengine cycles when no fbox_name/thread_num is specified. Number of Microengine instructions when fbox_name/thread_num is specified. A count of -1 means go indefinitely.

**Example:**

```
//Create a log file titled Test.log
log Test.log
//Execute initialization commands
chip// chip command
mem_init sdram 1m// mem_init command (sdram initialize)
mem_init sram 1m// mem_init command (sram initialize)
init// init command
//Execute deposit commands
deposit sim.error_handle_mode = 3
dep sim.error_handle_mode = 3
dep/s f0.art.enable = 1
dep/s f1.art.enable = 0
dep/s f2.art.enable = 0
dep/s f3.art.enable = 0
dep/s f4.art.enable = 0
dep/s f5.art.enable = 0
dep/s f0.art.debug_p0 = 1
dep/s f0.art.debug_p1 = 1
dep/s f0.art.debug_p2 = 1
dep/s f0.art.debug_p3 = 1
dep/s f0.art.debug_p4 = 1
```

```
// Set abort ops in the last 4 stages of the pipe
// in order to avoid processing garbage microwords (these
// will be latched in the last 4 stages at the start of
// the clock edge
dep/s f0.ctl.p0_abort = 1
dep/s f0.ctl.p1_abort = 1
dep/s f0.ctl.p2_abort = 1
dep/s f0.ctl.p3_abort = 1
//load_uc command
load_uc f0.ctl.ustore forloop.list *
// go command
go f0/0 25
exit
```

## 6.4.18 goto

Run the Transactor until one of three conditions has been met, as defined by goto_spec.

A state called **sim.goto_and_ubreak_key_on_p1** determines whether you go to pipeline stage 1 or 3. A state of 0 corresponds to pipeline stage 3. A nonzero value corresponds to pipeline stage 0. The default value of the state is 0. More information on states is contained in Appendix A.

**Format:** `goto{/silent} goto_spec {, max_cycles}`

**`/silent`** If this qualifier is specified, debug information is suppressed during the simulation.

**`goto_spec`** The goto_spec parameter may appear in any of the following formats:

- Cycles_count. If goto_spec is a single number, the Transactor will simulate up to and including the specified Microengine cycle number.
- Microword_addr#. If goto_spec is a number followed immediately by the pound (**#**) character, the Transactor will run until the specified microword numeric address has been reached in the P1 pipe stage of one of the specified contexts in one of the specified Microengines.
- Label#. If goto_spec is a microword label, the Transactor will run until the specified microword label has been reached in the P1 pipe stage of one of the specified contexts in one of the specified Microengines.

The microword_addr# and label# formats allow multiple entries (separated by spaces) in a single goto command. For multiple entries, the Transactor stops at the first address or label it encounters.

Where multiple threads are running at the same time, you can specify which thread(s) and which Microengine(s) the goto command should apply. Any thread(s) in any Microengine(s) can be specified. The following three model states are used by the goto command to enable threads and Microengines:

**`{chip_name}.art.fbox_mask`**
This state specifies one or more of the six Microengines and exists for every chip instantiated. The Microengines are specified by the low six bits where the LSB specifies Microengine 0 and the MSB specifies Microengine 5.

**`{chip_name}.art.ctx_mask`**
This state specifies one of 4 Microengine threads in the Microengine(s) enabled by the {chipname}.art.fbox_mask state. This state exists for

every chip instantiated. The threads are specified by the low four bits where the LSB specifies thread 0 and the MSB specifies thread 3.

**`{chip_name}.fx.art.ctx_mask`**
This state specifies one of 24 Microengine threads in the Microengine specified by fx, where x is the Microengine number. This state functions as a logical OR with the {chip_name}.art.ctx_mask state. Multiple Microengines and/or contexts can be simultaneously enabled, and if that is the case, the goto command stops at the first simulation time in which a match occurred.

**`max_cycles`** The Transactor runs until the goto point is reached or the number of cycles reaches max_cycles.

**Examples:**

```
> goto 34
> goto/silent 34#
> goto/s 34#

> goto label_1# label_2# label_3#
```

Set **goto** for Microengines 0, 2, and 3 in thread 0.

```
> dep/s art.fbox_mask = 7
```

Enables Microengine 0, 1,and 2.

```
> dep/s art.ctx_mask = 1
```

Enables all of thread 0.

Set **goto** for Microengine 0 thread 0:

```
> dep/s f0.art.ctx_mask = 1
```

Enables Microengine 0 thread 0.

## 6.4.19 help

Display helpful information about a topic or command.

**Format:**

```
help {topic_or_command}
```

**`topic_or_command`**
The topic or command about which you need more information. Valid entries are:

| | | | |
|---|---|---|---|
| CTRL-C | Overview | Command Interface | C-Interpretor |
| Model Initialization | Built-In C Functions | Simulation Switches | |
| #define | #elifdef | #else | #endif |
| #ifdef | #ifndef | #undef | @ |
| bank_analysis | chip | close | config |
| connect | debug | deposit | dev_test |
| examine | exit | fbox | go |
| go_clk_domain | goto | init | load_bin_file |
| load_list_file | load_uc | load_uof_file | log |
| mem_init | path | restore | return |
| save | set_clk_freq | sim_reset | statistics |
| time | trace | ubreak | uca |
| version | watch | | |

**Example:**

```
> help examine
```

Get help on the examine command.

### 6.4.20 #ifdef

The #ifdef, #ifndef, #elifdef, #else and #endif directives control execution of commands typed at the command line or in a script file. If the symbol following #ifdef is defined, the commands immediately following the #ifdef directive are executed up to the next #else, #elifdef, or #endif directive.

Each #ifdef/#ifndef directive must be matched by a closing #endif directive. Any number of #elifdef directives can appear between the #ifdef/#ifndef and #endif directives, but at most one #else directive is allowed. The #else directive, if present, must be the last directive before #endif.

The #ifdef, #ifndef, #elifdef, #else, and #endif directives can nest in other #ifdef/#ifndef directives. Each nested #else, #elifdef, or #endif directive belongs to the closest preceding #ifdef/#ifndef directive.

**Format:** `#ifdef macro_name`

### 6.4.21 #ifndef

The #ifdef, #ifndef, #elifdef, #else and #endif directives control execution of commands typed at the command line or in a script file. If the symbol following #ifndef is not defined, the commands immediately following the #ifndef directive are executed up to the next #else, #elifdef, or #endif directive.

Each #ifdef/#ifndef directive must be matched by a closing #endif directive. Any number of #elifdef directives can appear between the #ifdef/#ifndef and #endif directives, but at most one #else directive is allowed. The #else directive, if present, must be the last directive before #endif.

The #ifdef, #ifndef, #elifdef, #else, and #endif directives can nest in other #ifdef/#ifndef directives. Each nested #else, #elifdef, or #endif directive belongs to the closest preceding #ifdef/#ifndef directive.

**Format:** `#ifndef macro_name`

### 6.4.22 init

Initializes the model for simulation. Memory for all simulation states is allocated and initialized. All chip instantiations and SDRAM and SRAM instantiations must occur prior to performing this operation.

**Format:** `init`

**Example:** `> init`
Initialize the model.

### 6.4.23 load_bin_file

Loads the binary data from the specified file into the specified array state. If an index expression qualifies the array, the data will be loaded at the specified index; otherwise, the data will be loaded starting at location 1 (if the array state is SRAM or SDRAM) or location 0 in all other cases.

If the arrayed state to be loaded has a bit width which is not a multiple of 32 bits, this function assumes that memory storage for each state element is rounded up to an even number of 32-bit words. Thus, this function always assumes that the binary data contains an integer number of 32-bit words per array element.

**Format:** `load_bin_file array_name{[index_expression]} binary_file_name`

**Example:**

## 6.4.24 load_list_file

Loads the specified .list file (generated from the Assembler) into the specified memory or array state starting at the specified array index.

**Format:** `load_list_file ustore_state_name ucode_file_name starting_load_addr`

**Example:** This Transactor command loads forloop.list microcode starting at SRAM location 1:

```
load_list_file sram forloop.list 1
```

The Transactor responds back with:

```
Loaded 6 words from "forloop.list", into sram[1:6] (sram[0x1:0x6])
```

Since the assembled forloop microcode contains six microwords, it occupies six locations in SRAM.

## 6.4.25 load_uc

Loads the specified microcode file into the specified control store state name. Reads all microcode into the control store, but only validates the microcode corresponding to the list of page names specified.

**Format:** `load_uc ustore_state_name ucode_file_name {page_name_list}`

**`ustore_state_name`**
A microstore state name in the form {chip_name}.fx.ctl.ustore, where chip_name and fx are user specified.

**`code_file_name`**
Name of the microcode file to load.

**`page_name_list`**
Optional list of page names. An asterisk (*) entry in this field validates all pages.

**Example:**

```
load_uc f0.ctl.ustore ucode.list *
```

## 6.4.26 load_uof_file

Loads the specified uof file (generated from Linker) into the microcode control store. The uof file contains microcode for one or more microengines. If no chip_name is specified, then the uof file is loaded into the microcode control store for the unnamed/blank chip.

**Format:** load_uof_file uof_file_name {chip_name}

**`uof_file_name`** name of the uof file to be loaded.

**`chip_name`** If no chip_name is specified, then the uof file is loaded into the microcode control store for the unnamed/blank chip.

## 6.4.27 log

Opens a log file of the specified name to log commands, responses, or both. If file_name is not specified, displays all currently open log files. If neither commands nor responses are specified, both are assumed. A log file may be closed using the close command. All log files are automatically closed when the Transactor exits.

**Format:** `log {file_name} {commands} {responses}`

**`file_name`** The name of the log file to open.

**`commands`** The names of commands to log.

**`responses`** The names of responses to log.

**Example:**
```
> log myfile.txt
```
Open a log file.

## 6.4.28 mem_init

Instantiates an SRAM or SDRAM memory array of a specified size.

**Format:** `mem_init {/tag} mem_spec num_bytes`

**`/tag`** Enables each memory location to record when the location was written, what type of reference wrote it, which resource generated the reference, and at what microPC value. This option increases the virtual memory allocation of the memory structure by roughly a factor of 5, but is very useful when debugging.

**`mem_spec`** Designates the SRAM or SDRAM state name which have the forms: chip_name.sram and chip_name.sdram, respectively, where chip_name is the instantiated chip name.

**`num_bytes`** The k or m suffix can be included to designate kilobytes or megabytes, respectively. num_bytes specifies a number of bytes and not a number of memory words (there are 4 bytes per word for SRAM and 8 bytes per word for SDRAM).

**Example:**
```
> mem_init chip.sram 30m
```
Instantiate 30 megabytes of SRAM.

## 6.4.29 path

Analogous to the DOS **path** command. Allows the user to specify the search list of folders which is used to open files in the Transactor. Typing the command with no arguments displays the current path setting. Typing the command followed by a semicolon resets the path list to look only in the current folder area. Typing the command followed by a list of folder paths (separated by semicolons) specifies the list of folder paths that are searched in left to right order. The special keyword ***%path%*** specifies the previously existing search list.

**Format:** `path{;}{path_spec}{;path_spec}{...}`

**`path_spec`** The specification of the path used to open files in the Transactor.

**Example:** Example of the use of the **path** command in an IND file:

```
// clear current path
path .

// Set search path for Transactor two directories below
path ..\regression\ready_slave; ..\regression

// Display current path information
path


cmd("path ;");
cmd("path ..\regression\ready_slave; ..\regression");
```

## 6.4.30 restore

Restores the previously saved simulation state from the specified binary file.

**Format:** `rest{ore} file_name`

**`file_name`** The name of the file from which to restore a state.

**Example:**
```
> restore file.sav
```
Restore a previously saved simulation state from a file named file.sav.

## 6.4.31 return

When executed within a script file, return immediately halts execution of that script file even if more commands exist in the script. return has no effect when executed from the command line.

**Format:** `return`

**Example:**
```
> return
```

## 6.4.32 save

Saves the current simulation state to a binary file of the specified name. This saved state can be restored by reading back the saved file using the **restore** command.

**Format:** `save file_name`

**`file_name`** Specifies the name of the file used to save a state.

**Example:**
```
> save file.sav
```
Save the current simulation state in a file named file.sav.

## 6.4.33 set_clk_freq

Sets the simulator to simulate the Core, FBus and PCI clock domains according to the relative clock periods derived from the specified clock frequencies. All three clock frequencies must be specified. Each value must be specified in MHz and must be greater than or equal to 1 MHz. The values can be expressed as decimal numbers. This command must be executed prior to model simulation; the clock frequencies cannot be changed once one or more cycles have been simulated.

**Format:** `set_clk_freq core_clk_freq_in_MHz fbus_clk_freq_in_MHz pci_clk_freq_in_MHz`

**Example:** `> set_clk_freq 166 66 66`

### 6.4.34 sim_reset

Resets the state of the model to the point just after the model was first initialized.

**Format:** `sim_reset`

**Example:** `> sim_reset`
Reset the state of the model.

### 6.4.35 statistics

Lists all built-in Transactor statistics available in current simulation configuration.

**Format:** `stat{istics}`

**Example:** `> stat`

**Output:**

```
The following statistics are available:

f0.stat.compute_cycles:  Breakdown of compute cycles for f0

...
```

### 6.4.36 time

Prints the current wallclock time.

**Format:** `time`

**Example:** `> time`

**Output:** `Current wallclock time:  Mon Jun 18 16:19:28 2001`

### 6.4.37 trace

Opens a binary file, a verilog ASCII file, or a vcd file of a specified **file_name** in order to log the values of the specified list of simulation states over time. If neither the /verilog or /vcd qualifiers are specified, a binary file in TEMPEST format is opened. The TEMPEST format can be read by the DINOTRACE waveform display tool.

A trace file may be closed using the close command. All trace files are automatically closed when the simulator is exited, or when the restore command is executed. Trace files are automatically reopened whenever a restore operation is executed, which restores simulator state to the point in time in which that trace was first opened.

**Format:** `trace {/verilog}{/vcd}{/dots_to_underscores/} file_name name_list`

| | |
|---|---|
| **/verilog** | Specifies that a Verilog stimulus file (an ASCII file) be opened as the log file. |
| **/vcd** | Specifies that a vcd file be opened as the log file. |
| **/dots_to_underscores/** | Specifies that dots in specified signal names be converted to underscore characters in the trace file. |
| **file_name** | Specifies the name of the file that contains the trace logging data. |
| **name_list** | A list of simulation states separated by blanks. If @ is prepended to a name, the name refers to a file from which additional names will be derived. The wildcard character (*) may be used as a shorthand method of specifying one or more state names. |

If a signal is an array, an index range specification must be applied to address particular elements within the array. The index range specification (delimited by square brackets) can be a single C numeric expression or two numeric expressions separated by a colon (:) to indicate the inclusive range formed between the two numbers.

**Example:**
```
> trace trace.log sdram[0:100]
```

## 6.4.38 ubreak

Sets or clears a list of breakpoints for one or more specific Microengine threads or displays current microcode breakpoints. When a thread has been enabled to respond to a breakpoint, the Transactor unconditionally stops when a Microengine program counter reaches the breakpoint. Multiple breakpoints can be specified in one ubreak instruction.

A state called **sim.goto_and_ubreak_key_on_p1** determines whether you break on pipeline stage 1 or 3. A state of 0 corresponds to pipeline stage 3. A nonzero value corresponds to pipeline stage 0. The default value of the state is 0. Appendix A contains more information on states.

**Format:**
```
ubreak fx.ctl.ustore {uaddr_or_label =
thread_mask}...{uaddr_or_label = thread_mask}
```

| | |
|---|---|
| **fx.ctl.ustore** | Name of one of the six Microengine control store states, where fx = f0 through f5. |
| **uaddr_or_label** | Microengine control store address in the form of either an absolute address (number#) or a label (label#). |
| **thread_mask** | A four bit value that represents one of the four Microengine threads.The LSB represents thread 0 and the MSB represents thread 3. Threads are enabled by specifying a 1 and cleared be specifying a 0. |

**Example 1:**
```
> ubreak f0.ctl.ustore 5#=0xf
Breakpoint at PC:  5 of f0.ctl.ustore for
threads:  0 1 2 3
```
Set a breakpoint at control store address 5 for Microengine 0 and enable all threads.

**Example 2:**
```
0> ubreak *.ctl.ustore 5#=0x0
No breakpoints currently enabled for
f0.ctl.ustore
```

Clear a breakpoint at control store address 5 for all Microengines and all threads.

**Example 3:**

```
> ubreak f0.ctl.ustore 5#=0xf 13#=0xf
Breakpoint at PC:  5 of f0.ctl.ustore for
threads:  0 1 2 3
Breakpoint at PC:  13 of f0.ctl.ustore for
threads:  0 1 2 3
```

Set multiple breakpoints in the control store for Microengine 0 and enable all threads.

## 6.4.39 uca

Assemble a specified microcode source file.

**Format:**

```
uca file_spec
```

**`file_spec`** The path and filename of the microcode source file.

**Example:**

```
> uca filename.uc
> uca e:\myfiles\filename.uc
```

## 6.4.40 #undef

Delete a previously defined preprocessor macro name.

**Format:**

```
#undef macro_name {macro_argument_1,
macro_argument_2, macro_argument_3,...}
macro_text
```

**`macro_name`** Name of macro to be deleted. If the macro_name has arguments associated with it, the arguments are substituted into the macro_text whenever the corresponding formal argument name matches a token of the macro_text.

**Example 1:**

```
#define g go
#undef g
```

The first line defines g as the go command macro. The second line undefines or deletes the go command macro.

## 6.4.41 version

Displays the number associated with this version of the Transactor.

**Format:**

```
version
```

**Example:**

```
>version
Copyright (c) 1998 Intel Corporation
SA-1200 Transactor Version (safe_xact): 2.1.88
Compiled on Nov  2 2001 at 01:11:22
```

### 6.4.42 watch

Defines a watch state of the specified name that looks for state transitions for each of the specified state names after each simulation event completes. If one or more states are found to have transitioned, then the specified C statement associated with the watch is executed; otherwise no activity takes place.

Watches are automatically enabled when defined. Watches can be disabled and reenabled by assigning a zero or nonzero value to the watch_name state (e.g., dep watch_name = 0 or watch_name = 0;).

**Format:**

```
watch (watch_name, state_name1, state_name2,...,
state_namen) C statement
```

**`watch_name`** Name of watch state.

**`state_name1, state_name2,..., state_namen`**
Names of states.

**`C statement`** C statement to be watched.

**Example:**

```
> watch (watch1, dc.odd_cycle) {
printf ("dc.odd_cycle=%d/n", dc.odd_cycle);
if (++iteration_count==100)
sim.halt=1; //halt the model after 100 passes
}
```

## 6.5 C Interpreter

The Transactor contains an integrated C interpreter that implements a subset of the ANSI C language definition. The C interpreter is a powerful tool for creating test scripts. The C code can be typed in at the command prompt or entered using command script files.

The C interpreter has the following characteristics:

- Handles if, if/else, while, for, break, continue, return, code blocks (i.e., a list of C statements delimited by brackets ({}), and most C expressions).
- Any simple model state (i.e., a signal, register, or array (not a FIFO)) is treated as a global 32-bit int when it appears in a C statement or expression. See Appendix A for more details on states.
- User-defined int variables can be created. All user-defined variables are saved and restored when the model states are saved and restored. (unsigned int, char, etc. are not currently supported).
- As in C, scoping of variables is implemented based on their definitions within brackets {...}. Variables defined outside of any {...} are defined as global variables and can also be examined and deposited by Transactor commands.
- Functions may be defined and they can be called in the same manner as ANSI C functions. Predefined C functions are also provided. User-defined C functions with variable argument lists are currently not supported.
- C struct definitions can be defined and instantiated in both SRAM and SDRAM.
- When C text is interpreted, it is assumed that the entire C expression is specified on a single line. Multiple line C text can be entered using continuation characters that instruct the

interpreter to wait for more input before interpreting the text. Continuation characters are of two types:

— A continuation caused by a backslash (\) at the end of a line, which instructs the interpreter to wait for a subsequent line to be input, or

— An implicit continuation, which always exists as long as one or more left brackets ({) are not terminated with corresponding right brackets (}).

- C statements or expressions are evaluated to a precision of 32 bits (the initialization of 64-bit data elements requires the use of the **deposit** Transactor command).

### 6.5.1 Macros

The C interpreter supports macros using the **#define** and **#undefine** commands. Macros allow a user to specify text that is automatically substituted into every command line before the line is interpreted. This function allows users to customize the command interface. Macros can be defined at the command line or within command script files. The two forms of the syntax are:

```
#define macro_name macro_text
#define macro_name [(macro_argument, ... , macro_argument)] macro_text
```

Macro_text is a series of tokens, such as keywords, constants, or complete statements. One or more space characters must separate a token string from an identifier. Space characters are not considered part of the substituted text, nor are any spaces following the last token of the text.

Macro_argument names appear in macro_text to mark the places where actual values are substituted. Each parameter name can appear more than once in the macro_text, and the names can appear in any order. The number of arguments in the call must match the number of parameters in the macro definition.

The second syntax form allows the creation of function-like macros. This form accepts an optional list of parameters that must appear in parentheses. References to the macro_name after the original definition replace each occurrence of macro_name [(macro_argument, ... , macro_argument)] with a version of the token string argument that has arguments substituted for formal parameters.

Commas separate the formal parameters in the list. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate identifier and the opening parenthesis. Line concatenation can be accomplished by using a backslash (\) before the newline character for long directives on multiple source lines. The scope of a formal parameter name extends to the new line that ends token-string.

When a macro has been defined in the second syntax form, subsequent instances of macro_name [(macro_argument, ... , macro_argument)] constitute a macro call. The actual arguments following an instance of macro_name in the source file are matched to the corresponding formal parameters in the macro definition. Each token in macro_text that is not preceded by a stringing (#), token pasting (##) operator, or by a ### operator, is replaced by the corresponding actual argument. Three optional preprocessor operators provide the following functions:

## Inserted between two tokens in the macro_text to concatenate them.

# Prepended onto a macro_text token, causing the following token to be quoted.

### Prepended, causing the following token to be dequoted if quote characters exist as the first and last characters of the token.

Any nested macros (macros in the macro_text) are expanded before the directive replaces the formal parameter.

## 6.5.2 Predefined C Functions

The Transactor supports the following predefined C functions.

| C Function | Definition | Reference |
|---|---|---|
| add_bus_client | Add new bus client to existing bus. | 6.5.2.1 |
| alloc_array | Allocates contiguous memory. | 6.5.2.2 |
| alloc_list | Creates a linked list in memory. | 6.5.2.3 |
| amba_add_req | Adds an AMBA request. | 6.5.2.4 |
| cmd | Executes Transactor commands. | 6.5.2.5 |
| env_var | Returns 1 if specified environment variable exists. | 6.5.2.6 |
| field | Returns a specified field from a model state. | 6.5.2.7 |
| field_insert | Inserts data into a state at a specified field. | 6.5.2.8 |
| fprintf | Analogous to C fprintf. | 6.5.2.9 |
| gui | Returns 1 if Workbench is enabled. | 6.5.2.10 |
| pcit_add_req | Queues up a PCIT request. | 6.5.2.11 |
| pcit_set_slave_addr | Set the address range for a PCIT slave. | 6.5.2.12 |
| pcit_slave_read | Read a PCIT slave's memory. | 6.5.2.13 |
| pcit_slave_read | Write a PCIT slave's memory. | 6.5.2.14 |
| printf | Prints to stdio. | 6.5.2.15 |
| rand | Returns random integer. | 6.5.2.16 |
| rec_error | Analogous to C fprintf, but also registers simulation error. | 6.5.2.17 |
| seed | Returns current seed for random number generation. | 6.5.2.18 |
| set_bus_validity_checks | Sets validity checks for a named bus state object. | 6.5.2.19 |
| srand | Sets the seed for random number generation. | 6.5.2.20 |
| state_type | Returns numerically specified state type for defined states. | 6.5.2.21 |
| system | Implements the C system function. | 6.5.2.22 |
| uaddr | Returns the address of a label. | 6.5.2.23 |
| uninitialized | Returns a 1 if the model has been initialized. | 6.5.2.24 |
| valid_array | Returns the number of valid elements in a specified array. | 6.5.2.25 |

### 6.5.2.1 add_bus_client(bus_name, client_name)

Adds a new bus client to an existing bus. The bus_name must be previously defined. The client_name must be new. Assigning a value to the bus client causes that value to be driven onto the bus in the following bus cycle. A **sim_reset** command (without the **/preserve** switch) automatically deletes any user-defined bus clients that were created by this command.

### 6.5.2.2 alloc_array(array_state_name, struct_name, array_length)

Allocates contiguous memory within the specified array (either SRAM or SDRAM) by instantiating an array of the predefined struct type for the specified array length. The function returns the address of the start of the array.

**Example:**

```
// This example shows how to create an array of size 0x20000 in SRAM.
// There are three steps:
// 1. Define a structure
// 2. Define a pointer to the structure
// 3. Create the array in memory and return the pointer to the array
// Step 1
struct my_array \
    {
    int data = 0;
    };
// Step 2
struct my_array:sram *my_array_bass_addr;
// Step 3
my_array_base_addr = alloc_array(sram, my_array, 0x20000);
```

### 6.5.2.3 alloc_list(array_state_name, struct_name, list_length, flink_spec, blink_spec)

Allocates contiguous memory within the specified array (either SRAM or SDRAM) by instantiating a linked list of the predefined struct type to form a list of the specified length. The flink_spec (forward-link ptr) is used to specify the member name of the struct that acts as a pointer to the same struct type. The blink_spec (backward-link ptr) is used in an analogous fashion to specify the backward pointer for doubly linked lists. If no blink_spec is specified, a singly linked list is created. If only one member within the specified struct is defined as a pointer to the same struct, then the flink_spec need not be specified either, because the flink ptr can be determined unambiguously. The function returns the address of the start of the list.

**Example:**

```
// This example shows to create a linked list with three elements in
// SRAM. There are three steps
// 1. define a structure
// 2. define a pointer to the structure
// 3. create the linked list in memory and return a pointer to the
//    first element.
// Step 1
struct llist  \
{
    struct llist *pnext;
    int val;
};
// Step 2
struct llist:sram *pllist;
// Step 3
pllist = alloc_list( sram, llist, 3 );
```

### 6.5.2.4 amba_add_req(chip_name, write, addr, size, burst_size, num_prior_blank_cycles [, data...])

Adds an AMBA request to the AMBA Transactor command queue. The **write** parameter should be set to 1 to indicate a write operation, or 0 otherwise. The **addr** parameter is the byte address where the operation will occur. The **size** parameter indicates the size of the transaction and take on one of the values: 1, 2, or 4. The **burst_size** parameter should take on values from 1 to 8. The **num_prior_blank_cycles** parameter indicates the minimum number of cycles the AMBA bus should be idle since the previous transaction before this transaction starts. For read operations, no data is specified. For write operations, there should be as much data specified as the burst size.

### 6.5.2.5 cmd(quoted_cmd_string)

Executes the quoted_cmd_string as a Transactor command. This allows Transactor commands to be embedded inside C commands.

**Example:**

```
cmd("dep fbi.csr[32] = 0xffffffff");
```

### 6.5.2.6 env_var(char *environment_variable)

Returns 1 if the specified environment variable exists; otherwise it returns 0.

### 6.5.2.7 field(state_name, field_msb, field_lsb)

Returns the specified field within a state. The state_name can be any legal Transactor state and the bit field is specified by the field_msb and field_lsb arguments.

**Example:**

```
// This example deposits a zero into the RCV_RDY_LO register bit 5 and
// reads it back. Then it deposits a one into the RCV_RDY_LO register
// bit 5 and reads it back.
printf("depositing 0 now \n");
cmd("dep fbi.csr[32]<5:5> = 0");
printf("looking at it now using the field function \n");
field(fbi.csr[32], 5, 5);
printf("depositing 1 now \n");
cmd("dep fbi.csr[32]<5:5> = 1");
printf("looking at it now using the field function \n");
field(fbi.csr[32], 5, 5);
```

### 6.5.2.8 field_insert(state_name, insertion_data, field_msb, field_lsb)

Inserts the insertion_data into the specified state at the specified bit field. The insertion_data must be less than or equal to 32 bits. The function returns 1 if successful, otherwise 0.

### 6.5.2.9 fprintf(file_name, fmt, ...)

Analogous to C **fprintf** except that the first argument is a file name, not a file pointer. If the log file corresponding to the file name was not previously open, it is automatically opened by this call.

### 6.5.2.10 gui()

Returns 1 if the IXP1200 Developer Workbench is enabled in this simulation. Otherwise, returns 0.

### 6.5.2.11 pcit_add_req( slave_nbr, cmd, address, byte, burst, delay, data...)

Queue up a PCIT request. CMD is a PCI command, BYTE are the byte enables (active low), BURST is the size DATA is only supplied for write operations.

### 6.5.2.12 pcit_set_slave_addr( slave_nbr, start, len)

Set the address range for a PCIT slave. SLAVE_NBR indicates which slave is to be selected:

1..2. START is the start address in bytes, and LEN is the size of the region. The slave will respond to transactions that fall within this range for BOTH memory and IO space transactions. Note that while normal PCI devices must have address range lengths that are powers of 2, this is not required by the PCIT.

#### 6.5.2.13 pcit_slave_read( slave_nbr, io_space, address)

Read a PCIT slave's memory. SLAVE_NBR indicates which slave is to be selected 1..2, io_space is a boolean indicating that IO-space should be read. If it is false than MEM-space is read. ADDRESS is the PCI address (including base-registers) to be read.

#### 6.5.2.14 pcit_slave_write( slave_nbr, io_space, address, data, byte_en)

Write a PCIT slave's memory. SLAVE_NBR indicates which slave is to be selected 1..2, io_space is a boolean indicating that IO-space should be read. If it is false than MEM-space is read. ADDRESS is the PCI address (including base-registers) to be written. DATA is the data to be written, and BYTE_EN is an optional byte enable. If it is not specified, then 0xF is assumed.

#### 6.5.2.15 printf(const char *format, ...)

The **printf** function is analogous to **printf** in C.

**Example:**

```
// This example uses printf to report an error if the Microengine
// register ex4_result_0 is not equal to 0
if (f0.reg.ex4_result_0  != 0) {
        printf ("ERROR: alu-ex.uc\n");
    printf ("ERROR: Example 4 failed.\n");
    printf ("ERROR - expected: 0\n");
    printf ("ERROR - returned: %p\n",f0.reg.ex4_result_0);
}
```

#### 6.5.2.16 rand(bound1, bound2)

Returns a random integer between the two specified bounds, inclusive. If **bound2** is not specified, it defaults to 0. If neither bound is specified, a random 32-bit result is returned.

#### 6.5.2.17 rec_error(fmt, ...)

Analogous to **printf** function, but also registers a simulation error.

#### 6.5.2.18 seed()

Returns the current seed for random number generation.

#### 6.5.2.19 set_bus_validity_checks(bus, min_dead_cyc_wrn_thrsh, min_dead_cyc_err_thrsh max_dead_cyc_wrn_thrsh, max_dead_cyc_err_thrsh)

Sets the validity checks for the named bus state object. The min_dead_cyc threshold is the least number of dead cycles (cycles not being driven) required between two different sources driving the bus. The wrn_thrsh is the threshold for warnings, and the err_thrsh is the threshold for errors. The

max_dead_cyc threshold is the maximium number of dead cycles that can occur in a row. Values of -1 disable the given check. Each bus client is considered a different source. The verilog model and assignments directly from .ind files constitute another, single source.

### 6.5.2.20 srand(number)

Sets the seed for random number generation.

### 6.5.2.21 state_type(var_name)

Returns the numerically specified state type if the specified state name was previously defined. It returns 0 if the state is undefined. The argument may be either a quoted or unquoted name. Numeric values for the state types are as follows:

| | |
|---|---|
| Register | 1 |
| Signal | 2 |
| Verilog Signal | 3 |
| Verilog Wide Signal | 4 |
| Longword Array | 5 |
| Verilog Array | 6 |
| Verilog Wide Signal Array | 7 |
| Microcode Register | 8 |
| Model Artifact | 9 |
| String Pointer | 10 |
| Wide Signal | 12 |
| FIFO | 13 |
| Array | 14 |
| Memory | 15 |
| Control Store | 16 |
| CAM | 17 |
| Bus | 18 |
| Bus Client | 19 |
| Delay Element | 20 |
| Struct | 21 |
| Statistic | 22 |
| User-defined Variable | 23 |
| Watch | 24 |
| User-defined C Function | 25 |
| Macro | 26 |

### 6.5.2.22 system(char *shell_cmd)

Implements the C **system** function which passes a shell command to the OS or executing shell. The return status resulting from the execution of the command is returned by this function.

### 6.5.2.23 uaddr(control_store_state, label_name)

Returns the address of the specified label name within the control store of the Microengine that is specified by the control_store_state.

**Example:**

```
//This example uses uaddr to detect if the program reaches the label
// ethernet#. If this occurs, the address will be known during the
// Microengine execution pipeline stage 1. (During stage 0 the
// Microengine is fetching the next instruction). The f0.ctl.p1_abort
// state is inspected to ensure that the instruction at the label
// ethernet# is not aborted. This condition is noted by setting status
// equal to 0.
watch(w, sim.time) {
    if ((uaddr(f0.ctl.ustore, "ethernet#") == f0.art.p1_uaddr)
        && ! f0.ctl.p1_abort){
        status = 0;
        }
}
```

### 6.5.2.24 uninitialized()

Returns a value of 1 if the model is not initialized. Otherwise it returns 0. Certain predefined states act as simulation switches or parameters that you can set. All such states exist under the sim hierarchy. The following is a description of these states.

**Example:**

```
// ************************************************************
// This is the model setup and initialization section. The model is
// initialization includes:
// 1. one IXP1200 with a NULL chipname- chip
// 2. 1MB of SRAM memory-mem_init sram 1m
// 3. 1MB of SDRAM memory-mem_init sdram 1m
// 4. initialize the entire system -init

// Note that the chip and memory are initialized before the model. The
// If statement is not required to initialize the model. It's purpose
// is to allow this file to be re-loaded multiple times during a
// session. If this statement were not included the Transactor would
// report an error indicating that the chip and the memory can not be
// instantiated after the model is initialized. The If statement uses
// the un-initialized function to determine that if the model has been
// initialized. The sim_reset command is used to reset the model.
*********************************************************************
if ( uninitialized() )
    {
    // instantiate a chip using the chip command
    cmd( "chip" );
    // instantiate some memory using the mem_init command
    cmd( "mem_init sdram 1m" );
    cmd( "mem_init sram 1m" );
    // initialize the chip using the init command
    cmd( "init" );
    }
// reset all post-init simulation state
// using the sim_reset command
else cmd( "sim_reset" );
//load microcode into Microengine 0.
load_uc f0.ctl.ustore llist.list *
```

#### 6.5.2.25 valid_elements(array_state_name)

Returns the number of valid elements in the specified array.

## 6.6 Debugging

The IXP1200 Transactor supports a number of powerful debugging features that are provided through the Transactor commands, C statements, and model states. This section describes some of these features and will provide examples of how to use them. Debugging features include:

- Debug summary information reporting
- Memory tags
- Watch functions
- Breakpoints
- Saving and restoring the model state
- Logging the simulation session

### 6.6.1 Reporting of Debugging Information

Debugging information can be displayed when the debug command is executed or upon completion of a go or goto Transactor command. The debug information is displayed based on the values of the following two states.

- unit.art.debug
- fx.art.debug_px

These states are set and viewed using the Transactor examine and deposit commands. See Appendix A for further information on these states.

```
unit.art.debug (where unit = fbi, sc, dc, or f0-f5)
```

This state enables a summary of debug information to be displayed for the FBI, SRAM, SDRAM, and Microengines. Depositing a value of 0 (disabled) through 4 (most information) to this model state sets the level of information that is displayed, as shown in the following tables:

**Table 6-3. State Setting Descriptions for fx.art.debug**

| fx.art.debug State Setting | Description |
|---|---|
| 0 | Disabled. |
| 1 | View execution pipeline stage 3 (ALU output). |
| 2 | View all 5 execution pipeline stages. |
| 3 | View all 5 execution pipeline stages and thread state information. |

**Table 6-4. State Setting Descriptions for *.art.debug**

| sc.art.debug<br>dc.art.debug<br>fbi.art.debug State Setting | Description |
|---|---|
| 0 | Disabled. |
| 1 | Enabled. |

**fx.art.debug_px**
(where fx = f0 through f5, and debug_px = debug_p0 through debug_p4)

When fx.art.debug is enabled to display execution pipeline stage information, the fx.art.debug_px state disables the display of the specified Microengine execution pipeline stage information. For example, when f0.art.debug is set to 4, the Microengine execution pipeline stage information for each stage will be displayed. If f0.art.debug_p3 is set to 0 then only stages 0,1,2, and 4 will be displayed. Depositing a value of 0 disables the display.

The work performed during each pipeline stage is summarized below:

**Table 6-5. Pipeline Stage Work**

| Stage | Work Performed |
|---|---|
| 0 | Read the instruction from the control store. |
| 1 | Decode the instruction. |
| 2 | Look up the operands. |
| 3 | Perform the ALU/shifter operations. |
| 4 | Write result to the destination register. |

Depositing a value of 0 (disabled) or 1 (enabled) to this state determines whether the information for the execution pipeline stage is displayed. The format of the information displayed is shown in the sections that follow.

## 6.6.2 Common Execution Pipeline Stage Format

Each execution pipeline displays the same information on the first three lines.

```
====P4 fbox stage (write result) for f0. in ctx: 0 at fbox cycle 63====
uword addr = 12
    alu_shf_ri[loop_no, loop_no, -, 1, 0]      ; decrement link_no
```

Where:

Line 1 shows the Microengine execution pipeline stage number (P0 through P4), a description of the pipeline execution stage (write result), the Microengine executing this instruction (f0 through f5), the thread/context number executing this instruction (ctx: 0), and the Microengine execution cycle (fbox cycle 63).

Line 2 shows the location of this instruction in the control store (uword addr = 12). If the instruction is aborted due to a branch or other condition, ***ABORTED_UWORD*** will appear on this line. If the Microengine is stalled because the Microengine has more than two external references waiting to be issued to the appropriate function unit ***STALLED UWORD *** will appear.

Line 3 shows the instruction in this execution pipeline stage including comments.

The formats described in the sections that follow are displayed for each execution pipeline stage.

### 6.6.2.1 Execution Pipeline Stage 4 - Write the Result

This is the complete display for execution pipeline stage 4.

```
====P4 fbox stage (write result) for f0. in ctx: 0 at fbox cycle 63====
uword addr = 12
    alu_shf_ri[loop_no, loop_no, -, 1, 0]      ; decrement link_no
W_BUS:  00000002
operand write:  GPR_A[0]=00000002
```

Where:
Lines 1 through 3: Common execution pipeline stage information.

Line 4 shows the output of the ALU (W_BUS = 00000002).

Line 5 is displayed if the output of the ALU has a destination. It displays where the result is being written (GPR_A[0]) and the value (00000002). The value should be the same as W-BUS in line 4.

### 6.6.2.2 Execution Pipeline Stage 3 - Perform ALU Shift, Decode Instruction

This is the complete display for execution pipeline stage 3.

```
====P3 fbox stage (ALU/shifter) for f0. in ctx: 0 at fbox cycle 63====
uword addr = 13
    br!=0_[loop_rl#], defer[2]
A operand: 00000000; pre-shift B operand: 00000007; B operand: 00000007
shifter op: right shift 0; ALU op: ALU_OP_A_AND_B; ALU result:00000000;
old_ccs: !=0:>=0:carry
```

Where:
Lines 1 through 3:  Common execution pipeline stage information.

Line 4 shows the A, B and preshift B operand values. If no shift operation is performed the B and preshift B operand values will be the same.

Lines 5 and 6 shows the ALU and shifter operation that were performed, the result of this operation and the condition code status after the operation. If the condition code changes old_ccs will be displayed, otherwise new_ccs is displayed. The possible condition code values are:

**Table 6-6. Condition Code Meanings**

| Condition Code | Meaning |
|---|---|
| !=0 | Not equal to zero. |
| =0 | Equal to zero. |
| >=0 | Greater than or equal to zero. |
| <0 | Less than zero. |
| carry | Carry. |
| no_carry | No carry. |

#### 6.6.2.3 Execution Pipeline Stage 2 - Look Up Operands

This is the complete display for execution pipeline stage 2.

```
====P2 fbox stage (operand lookup) for f0. in ctx: 0 at fbox cycle 63==
uword addr = 14
    new_head#:
        nop
A operand:  00000000  (selected from xreg data);
B operand:  00000000  (selected from immed data);
```

Where:
Lines 1 through 4: Common execution pipeline stage information.

Lines 5 and 6 show the operand values that were fetched per instruction.

#### 6.6.2.4 Execution Pipeline Stage 1 - Decode Instruction

This is the complete display for execution pipeline stage 1.

```
===P1 fbox stage (initial decode) for f0. in ctx: 0 at fbox cycle 63===
uword addr = 15
    ld_field_a[headptr, 1111, $ptr, 0]
```

Where:
Lines 1 through 3: Common execution pipeline stage information

#### 6.6.2.5 Execution Pipeline Stage 0 - Read Instruction

```
====P0 fbox stage (ustore lookup) for f0. in ctx: 0 at fbox cycle 63===
ustore addr = 10 ==> A018108A
    loop_rl#:
        sram[pop,  $ptr, temp, temp1, 0], ctx_swap
```

Where:
Lines 1 through 3: Common execution pipeline stage information.

### 6.6.3 Memory Tagging

The Transactor provides the ability to tag SRAM or SDRAM memory with information about which location was written, what type of reference wrote it, which resource generated the reference, and at what Microengine program counter value. Memory tagging is enabled when memory is initialized using the mem_init command.

### 6.6.4 Watch

The watch function is a powerful tool that executes code upon the transition of one or more user defined states. One purpose of the watch function is to observe states and display information on transitions that occur upon the occurrence of some combination of conditions. It is also used to halt simulation or deposit data into states when some combination of conditions occurs. The watch function is invoked by using the watch command (see the watch command description).

The watch function can be defined in a command script file and run using @command_script_filename.

### 6.6.5 Breakpoints

Breakpoints are supported by the Transactor using the ubreak command. A breakpoint causes the simulator to stop unconditionally when a thread that is enabled to respond to that breakpoint reaches the specified Microengine program counter address. Breakpoints can be set or cleared on a thread-by-thread basis. Any number of breakpoints can be set or cleared by specifying one of the six Microengine control store states followed by a microcode address in the form of either number# or label#, followed by an equal sign (=), followed by a thread mask value. The thread mask value is a four bit value where the LSB specifies thread 0 and the MSB specifies thread 3.

### 6.6.6 Save and Restore

The save and restore commands are used to save and restore the state of a model and are used frequently in connection with debugging. See the save and restore command descriptions for more information.

### 6.6.7 Logging A Simulation Session

A log file can be created that contains information displayed during a simulation session. Log files are opened using the log Transactor command and are closed using the close Transactor command. See these command descriptions for more information.

## 6.7 Transactor Command Script Files

The Transactor can run command scripts that contain Transactor commands and C commands supported by the Transactor's C interpreter. The following rules apply to Transactor command scripts.

- C commands that require more than one line must terminate each line with a continuation character to instructs the interpreter to wait for more input before interpreting the text. Continuation characters are of two types:
  - — A backslash (\) at the end of a line instructs the interpreter to wait for the subsequent line to be input.
  - — An implicit continuation, always exists as long as one or more left brackets ({) are not terminated with corresponding right brackets (}).
- Comments can be defined two ways.
  - — ANSI C - starting after a slash-asterisk (/*) and ending before an asterisk-slash (*/).
  - — C++ - comment starts after a double slash (//) and runs to the end of the current line.

There are two ways to execute a command script file:

1. A command script file can be executed at the command line interface using the @script_filename Transactor command.
2. A single command script file can run automatically when the Transactor is executed using the -i switch (e.g., safe_exact -i script_filename).

The safe_exact -i script_filename format can be used to create automated test script. An automated test script would contain a series of command using this format. The last command in each script_filename would contain the Transactor exit command so that the Transactor exits after the command script completes.

Command scripts can be aborted by placing the return command into a command script.

### 6.7.1 Example Command Script File

The following is an example Transactor command script file.

```
// This is an init file
//*********************************************************************
// This is the model setup and initialization section. The model
// initialization includes:
// 1. One IXP1200 with a NULL chipname- chip
// 2. 1MB of SRAM memory- mem_init sram 1m
// 3. 1MB of SDRAM memory- mem_init sdram 1m
// 4. Initialize the entire system - init
//
// Note that the chip and memory are initialized before the model. The
// If statement is not required to initialize the model. Its purpose
// is to allow this file to be reloaded multiple times during a
// session. If this statement were not included the Transactor would
// report an error indicating that the chip and the memory can not be
// instantiated after the model is initialized. The If statement uses
// the uninitialized function to determine whether the model has been
// initialized. The sim_reset command is used to reset the model.
//*********************************************************************
{
if ( uninitialized() )
    {
    // instantiate a chip
    cmd( "chip" );
    // instantiate some memory
    cmd( "mem_init sdram 1m" );
    cmd( "mem_init/tag sram 1m" );
    // Initialize the chip
    cmd( "init" );
    }
else cmd( "sim_reset" );    // reset all post-init simulation state
}
//*********************************************************************
// This section sets up the debug reporting states.
//*********************************************************************
dep dc.art.debug = 0            //do not view SDRAM debug info
dep sc.art.debug = 0            //do not view SRAM debug info
dep f0.art.debug = 2            //view F0 debug info
dep f0.art.debug_p0 = 1         //view all 5 pipeline stages
dep f0.art.debug_p1 = 1
dep f0.art.debug_p2 = 1
dep f0.art.debug_p3 = 1
dep f0.art.debug_p4 = 1
//*********************************************************************
// This section turns off unused Microengines
//*********************************************************************
dep f1.art.enable = 0   // only Microengine 0 will run
dep f2.art.enable = 0// 0 = off
dep f3.art.enable = 0// 1 = on
dep f4.art.enable = 0
dep f5.art.enable = 0
//*********************************************************************
// This section turns off unused Microengines
//*********************************************************************
```

```
dep art.fbox_mask = 1           // Turn on f0  (binary representation)
// This is a six bit mask where
// lsb = uengine 0, msb = uengine 5
//*********************************************************************
// This section turn on the Microengines threads
//*********************************************************************
dep art.ctx_mask = 0xf          // Turn on all threads
// This is a four bit mask where
// lsb = thread 0, msb = thread 3
//*********************************************************************
// This section is only temporary. This will not be required when the
// circuit is implemented.
// 0, msb = uengine 5  0, msb = uengine 5  0, msb = uengine 5  0,
// msb = uengine 5
// This section sets abort operands in the last 4 stages of the
// execution pipe in order to avoid processing garbage uwords (these
// will be latched in the last 4 stages at the start of the clock edge.
//*********************************************************************
dep f0.ctl.p0_abort = 1
dep f0.ctl.p1_abort = 1
dep f0.ctl.p2_abort = 1
dep f0.ctl.p3_abort = 1
//*********************************************************************
// This section loads the microcode into the program store
//*********************************************************************
    load_uc f0.ctl.ustore llist.list *
//*********************************************************************
// This section is specific to the microcode program. It shows how a
// linked lists can be placed into SRAM memory.
// The steps are as follows:
//      1. Create a structure for the elements in the list.
//      2. Define a pointer to the structure
//      3. Create the list in memory
//      4. Write the head of the list pointer to the push pop register
//      5. Initialize memory with data
//*********************************************************************
// Note that the following 5 lines demonstrate the two forms of
// continuation characters. The "\" character after the first line
// indicates that the following line is a continuation of the first
// line. Everything within brackets "{}" are also interpeted as a
// continuation.
struct llist  \
{   struct llist *pnext;// Step 1
    int val;
};
struct llist:sram *pllist;// Step 2
pllist = alloc_list( sram, llist, 3 );// Step 3
// The alloc_list function is used to create three elements
// of type llist in sram. The return value is the pointer
// to the first element in the list. Note that the first
// longword in each element is a pointer to the next element
// in the list. The last list has a NUL pointer
dep/s sc.push_pop_regs[0] = pllist// Step 4
// Write the pointer to the first element in the list to the SRAM
// unit's push-pop register
// initialize the values in the array// Step 5
int i;  // initialize memory with data
for (i=1; i<=3; i++) {// This example simply writes
    pllist ->val = (i*10);// 10, 20, and 30 into the list
    pllist++;
};
//*********************************************************************
// This section deposits the head of the linked list into a
// Microengine GPR for each thread.
//*********************************************************************
dep f0.reg.headptr_0=pllist
```

```
dep f0.reg.headptr_1=pllist
dep f0.reg.headptr_2=pllist
dep f0.reg.headptr_3=pllist
//*********************************************************************
// This section
//*********************************************************************
dep/multi f*.ctl.thread_trigger_mask* = 1 // turn on all threads
dep/multi f*.ctl.thread_trigger_mask*0 = 0// turn off 0 thread (which
           //  is on by default)
```

## 6.8 Enabling the StrongARM Core

This section describes how to enable and disable the StrongARM Core model in the Transactor. You may want to disable the StrongARM core for the following reasons:

- You are not executing ARM code and want to improve Transactor performance by disabling the StrongARM Core.
- You prefer to use the AMBA Transactor rather than the StrongARM Core model.

The IXP1200 Developers Workbench automatically performs the necessary steps required to enable or disable the StrongARM Core, but users that choose to use the command line interface must manually perform the steps that follow. To make this easier, it is recommended that you create separate script files to enable and disable the StrongARM Core. Then you can simply execute the script files.

The following states are associated with enabling and accessing the StrongARM Core:

| | |
|---|---|
| art.pci_enable | Enable PCI registers (and eventually the PCI bus) |
| art.kbox_enable | Enable StrongARM Core |
| art.amba_xact_enable | Enable AMBA Transactor |
| art.resets_connected_to_reset_csr | Connect resets to SA1200_RESET register |

To disable the StrongARM Core model in the Transactor set the states as follows (these are the default settings):

| | |
|---|---|
| art.pci_enable | 0 |
| art.kbox_enable | 0 |
| art.amba_xact_enable | 1 |
| art.resets_connected_to_reset_csr | 0 |

To run the Transactor with the StrongARM Core functioning, set the states as follows:

| | |
|---|---|
| art.pci_enable | 1 |
| art.kbox_enable | 1 |
| art.amba_xact_enable | 0 |
| art.resets_connected_to_reset_csr | 1 |

Do not set both the **art.kbox_enable** and **art.amba_xact_enable** states at the same time. Results will be unpredictable.

The state **art.resets_connected_to_reset_csr** when set to 1 connects the SA1200_RESET register in the PCI unit to the reset lines of the other functional units. Refer to the *IXP1200 Network Processor Programmer's Reference* for more information on the SA1200_RESET register. This state must be set for the SA1200_RESET register to work properly, but it is off by default for compatibility with .ind files written for previous versions of the Transactor. Set this state to 1 when writing new .ind files.

The SA1200_RESET register is known in the Transactor as **p.sa1200_reset_reg**.

### Example

To enable the StrongARM Core, the .ind file (after the initialization part) might start with the following:

```
> art.pci_enable = 1;
> art.kbox_enable = 1;
> art.amba_xact_enable = 0;
> art.resets_connected_to_reset_csr = 1;

> p.sa1200_reset_reg = 0xFFFFFFFF; //Resets the whole IXP1200 chip.
> go 8
> p.sa1200_reset_reg = 0x1FFEFFFF; //Brings SRAM, StrongARM Core, and FBus out
                                   //of reset.

> watch (ucode_watcher, c.r[6]) ucode_func(); //Look at register 6 to determine
                                              //when to load microcode.
> void ucode_func() {
>      // load ucode here
> }

> load_bin_file flash[0] your_ARM_binary_code_filename_goes_here
```

The StrongARM assembly source code might be structured as follows:

```
;; initialize processor, bring SDRAM out of reset, etc.
...
;; prepare Microengines to receive microcode (they are already in
;; reset) - clear context_enables

;; load ucode
MOV R6, #1
;; This doesn't really load the microcode in real hardware, but here
;; it fires off the watcher to load the microcode directly (and more quickly)

;; Initialize and bring Microengines out of reset
;; ...
;; Take Microengines out of reset
;; Set Microengine CTX_ENABLES register
;; ...
```

## 6.9 Simulation Switches

Several pre-defined simulation states exist that act as simulation switches/parameters for the user to set. All such states exist under the "sim." hierarchy.

The following is a description of the states:

| Switch | Section |
|---|---|
| sim.error_count | 6.9.1 |
| sim.error_handle_mode | 6.9.2 |
| sim.core_clk_cycle | 6.9.3 |
| sim.core_clk_freq | 6.9.4 |
| sim.core_clk_period | 6.9.5 |
| sim.fbus_clk_cycle | 6.9.6 |
| sim.fbus_clk_freq | 6.9.7 |
| sim.fbus_clk_period | 6.9.8 |
| sim.goto_and_ubreak_key_on_p1 | 6.9.9 |
| sim.halt | 6.9.10 |
| sim.pci_clk_cycle | 6.9.11 |
| sim.pci_clk_freq | 6.9.12 |
| sim.pci_clk_period | 6.9.13 |
| sim.post_sim_exec_order | 6.9.14 |
| sim.prune_prefix_comments | 6.9.15 |
| sim.silent | 6.9.16 |
| sim.time | 6.9.17 |

### 6.9.1 sim.error_count

Indicates the number of errors that were flagged during the session.

### 6.9.2 sim.error_handle_mode

Indicates what action should be taken when the error occurs.

The valid values are:

-1 Suppress and ignore error message.

0 Print the error but ignore its occurrance (i.e. the sim.error_count is not incremented and the simulation continues).

1 Print the error, increment sim.error_count and halt simulation if simulation is in progress.

2 Print the error, increment sim.error_count and halt simulation and command line/ script file execution any is in progress.

3 Print the error, increment sim.error_count, halt all execution and then exit the simulator. The default value is 2.

### 6.9.3 sim.core_clk_cycle

Indicates the current number of core clock cycles simulated.

### 6.9.4 sim.core_clk_freq

Indicates the core clock frequency (in MHz).

### 6.9.5 sim.core_clk_period

Indicates the number of simulation time units specified for a core clock cycle.

### 6.9.6 sim.fbus_clk_cycle

Indicates the current number of fbus clock cycles simulated.

### 6.9.7 sim.fbus_clk_freq

Indicates the fbus clock frequency (in MHz).

### 6.9.8 sim.fbus_clk_period

Indicates the number of simulation time units specified for an fbus clock cycle.

### 6.9.9 sim.goto_and_ubreak_key_on_p1

This state is relevant to the "goto", "ubreak" and "go 'fbox/ctx'" commands. It indicates whether to stop simulation by examining P1 uword state or P3 uword state. P3 state is desired in most cases. P1 state will stop simulation earlier in the pipeline, but may "falsely" stop because the P1 uword may become aborted in P2 or P3.

### 6.9.10 sim.halt

This state is normally 0. If the user sets it to 1, the model halts (if running) at the end of the current cycle. This is useful in watch statements when it is desired for the model to be halted when a specific condition has been met. Note that this state must be reset to 0 in order to continue running the model.

### 6.9.11 sim.pci_clk_cycle

Indicates the current number of pci clock cycles simulated.

### 6.9.12 sim.pci_clk_freq

Indicates the pci clock frequency (in MHz).

### 6.9.13 sim.pci_clk_period

Indicates the number of simulation time units specified for an pci clock cycle.

### 6.9.14 sim.post_sim_exec_order

Indicates the order in which watches, callbacks and the foreign_model_post_sim routine are called after each transactor simulation event is executed. Because watches, callbacks and foreign_model_post_sim offer the user 3 different mechanisms to either actively change state, and/or passively observe state, the possibility arises that the execution order of these mechanisms may cause incorrect behavior. For example, performing passive reads (e.g. informational debug statements) prior to an active state change could cause incorrectly display debug data; or the order in which active state changes are evaluated could cause incorrect behavior.

This switch allows the execution order of these 3 mechanisms to be set based upon the user-defined application of these 3 mechanisms. Using the notation: W (watches), C (callbacks) and P (foreign_model_post_sim), the legal values for this switch are: 0: W-C-P, 1: W-P-C, 2: C-W-P, 3: C-P-W, 4: P-W-C, 5: P-C-W. The default value is W-C-P.

### 6.9.15 sim.prune_prefix_comments

This state is normally 0. When set to 1, ucode comment lines are pruned so that pure comment lines are not displayed--only lines with actual ucode descriptive text are displayed.

### 6.9.16 sim.silent

This state is normally 0. If the user sets it to 1, all debug information is suppressed.

### 6.9.17 sim.time

Represents current simulation time.

# Foreign Model Simulation Extensions 7

Foreign Model simulation extension is a useful tool for simulating external hardware devices, and, in development StrongARM core software. It acts as an extension to the capabilities of Transactor, which is a cycle and data accurate software model of IXP1200. Section 7.1 provides an overview of where foreign model simulation can be used. Section 7.2 describes how to integrate foreign models with the Transactor. Section 7.3 discusses the simulation of software modules running on the StrongARM Core. Details of simulating IX Bus devices are discussed in Section 7.4. Section 7.5 contains some sample code.

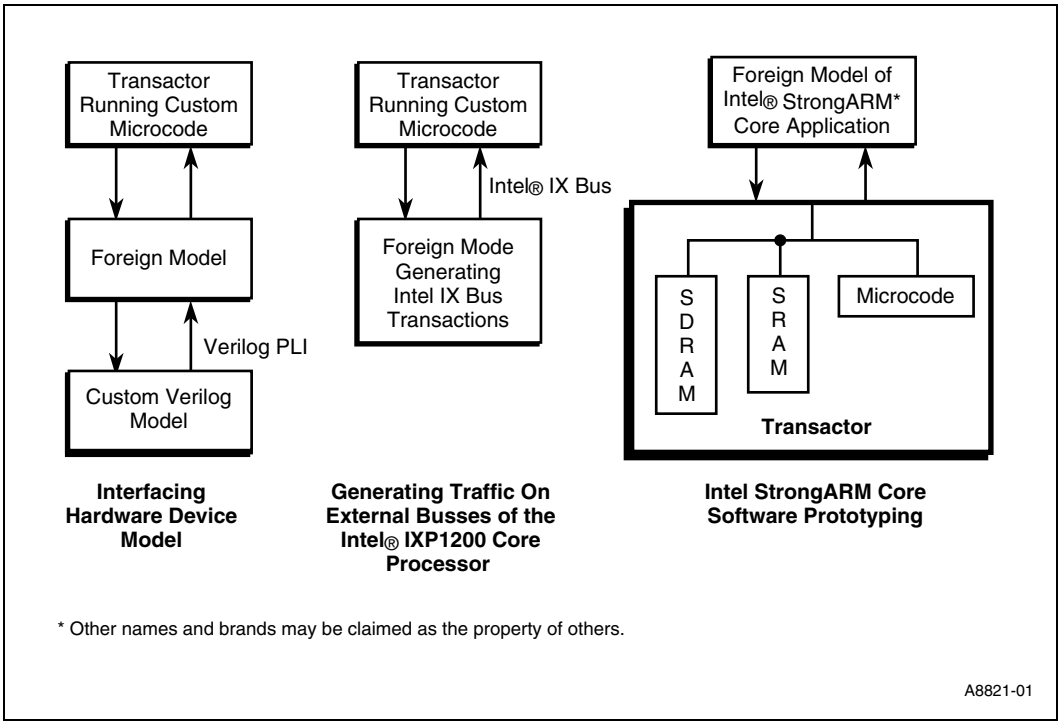
## 7.1 Overview

There are three reasons to use Foreign Model simulation:

1. To interface hardware device models,
2. To generate traffic on external busses of IXP1200, and,
3. To perform rapid prototyping of StrongARM Core software modules.

These three are illustrated in Figure 7-1.

**Figure 7-1. Three Uses of Foreign Models**




- **Interfacing hardware device models**. Foreign modeling allows integration of models of external hardware such as a MAC device or custom chip that is connected to the IX Bus. This

could be a software model of the external device or a design in a hardware simulator such as Verilog. This includes the ability to interface with hardware models (such as a Verilog model) which could reside on a remote machine.

- **Generating traffic on external busses**. Microengine software designers can use a foreign model to assist in the design and debugging of Microengine software modules by producing generic transactions of the IX Bus. This way, hardware residing on the IX Bus such as a MAC device or some custom chip can be simulated. Once the software is designed, the same foreign model interface can be used to produce the traffic typical for the application. This assists in estimating the performance of software.
- **StrongARM Software Module Prototyping**. The Foreign model interface can also be used to develop the software that will run on the StrongARM Core. Even though StrongARM software executes on a development machine, once it interfaces to the Transactor through the Transactor API, execution is cycle accurate. This reduces the simulation time and allows accurate verification of interactions between StrongARM and Microengine software.

## 7.2 Integrating Foreign Models with the Transactor

A foreign model provides a mechanism by which the software model of the IXP1200 (Transactor) can be extended to include additional software models of hardware that interfaces with the IXP1200. There are three different ways to integrate a foreign model with the Transactor.

- **Foreign Model Dynamic-link Library (DLL):** Used when the foreign model is running on the same Microsoft Windows system as the Transactor, and the user chooses to use the Developer's Workbench Graphical User Interface.
- **Integrated Executable:** Used when the foreign model is running on the same system as the Transactor, and the user chooses to use the command line interface rather than the Developers Workbench Graphical User Interface.
- **Remote Foreign Model Executable:** Used to interface a hardware simulator (VHDL or Verilog) running on Unix to the Transactor and the Developer's Workbench (which run on WindowsNT).

To integrate the foreign model, the Transactor calls a foreign model at five stages of simulation:

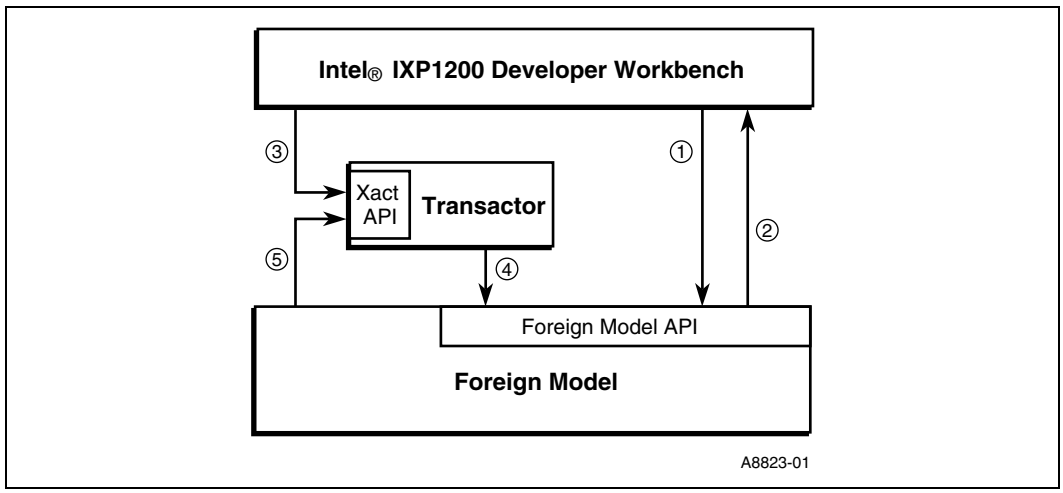
- Immediately after the Transactor has initialized the IXP1200 model,
- Before a simulation step occurs,
- After a simulation step has occurred,
- When the simulation is reset, and
- When the Transactor exits.

The foreign model must provide five foreign model functions corresponding to these five calls. The foreign model interacts with the Transactor through the Transactor API, described in the file xact.h.

## 7.2.1 Foreign Model Dynamic-Link Library (DLL)

This section describes how to create a dynamic-link library (DLL) for a foreign model simulation extension. The DLL is used in conjunction with the Transactor DLL when running the IXP1200 Developer Workbench. The Workbench, Transactor, and foreign model interact as shown in Figure 7-2.

**Figure 7-2. Workbench, Transactor, and Foreign Model Interaction**




1. The Workbench requests from the foreign model the pointers to foreign model functions through the foreign model API.
2. The foreign model replies with the function pointers.
3. The Workbench passes the pointers to the Transactor through the Transactor API.
4. As the Workbench controls the Transactor through the Transactor API, the Transactor notifies the foreign model whenever:
   a. The Workbench is initialized,
   b. Before a simulation step occurs (preSim),
   c. After a simulation step has completed (postSim),
   d. The simulation is reset (i.e., a sim_reset command is executed), and
   e. When the Workbench exits debug mode and releases the Transactor.
5. The foreign model interacts with the Transactor using the Transactor API.

The foreign model DLL must provide the exported function GetForeignModelFunctions(), that the Workbench calls to get the addresses of the five functions that the Transactor calls to interact with the foreign model. The Workbench registers these functions with the DLL version of the Transactor.

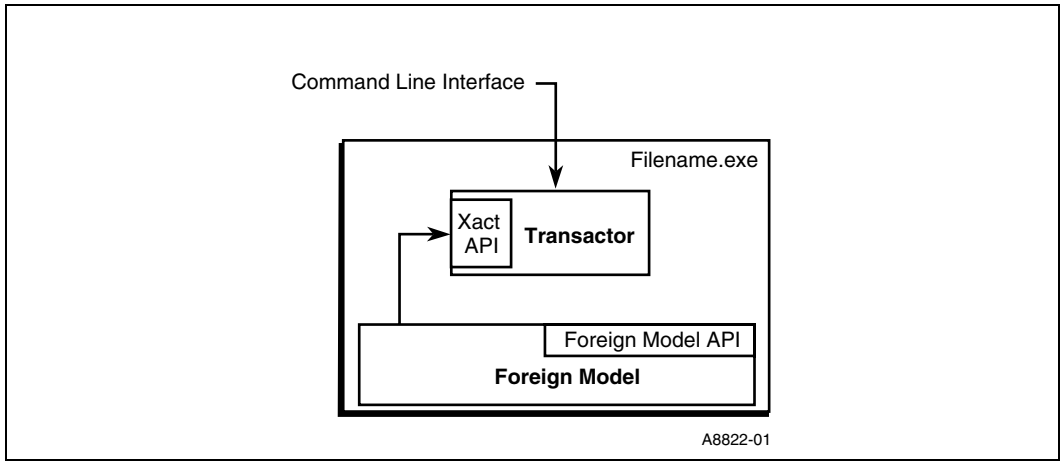
The foreign model DLL runs in conjunction with the DLL version of the Transactor, so xact_dll.h must be included in the foreign model DLL source files, if needed, and it must be linked against xact_dll.lib.

To specify that the DLL simulator is to be used as an extension to the Transactor model in the Workbench, select Debug > Simulator Options then select the Configuration tab. Select Local Simulation and check the Use simulator extension option. Finally, type the file name for your foreign Model DLL into the field provided, or browse for it by clicking on the '…' button.

## 7.2.2 Integrated Executable (Integrated Transactor and Foreign Model)

This section describes how to create an executable containing both the Transactor and a foreign model simulation extension. The foreign model and Transactor interact the same as the DLL except that the pointers to the foreign model API function calls are determined at compile time. The Transactor header file xact.h must be included in all files that call the Transactor API. Any object modules must be linked with xact_main.lib to create an executable. When the executable file is executed, the user can enter Transactor commands and any console commands that are defined in the foreign model using the command line interface (Figure 7-3).

**Figure 7-3. Command Line Interface**




## 7.2.3 Remote Foreign Model Executable

The Remote Foreign Model uses the Open Network Computing Remote Procedure Protocol (ONC RPC) to interface the IXP1200 Developer Workbench and Transactor DLL (executing on an WindowsNT system) to a foreign model executing on another remote system. The advantage to remote simulation is that the foreign model can execute on other platforms such as a Sun Workstation running Solaris.

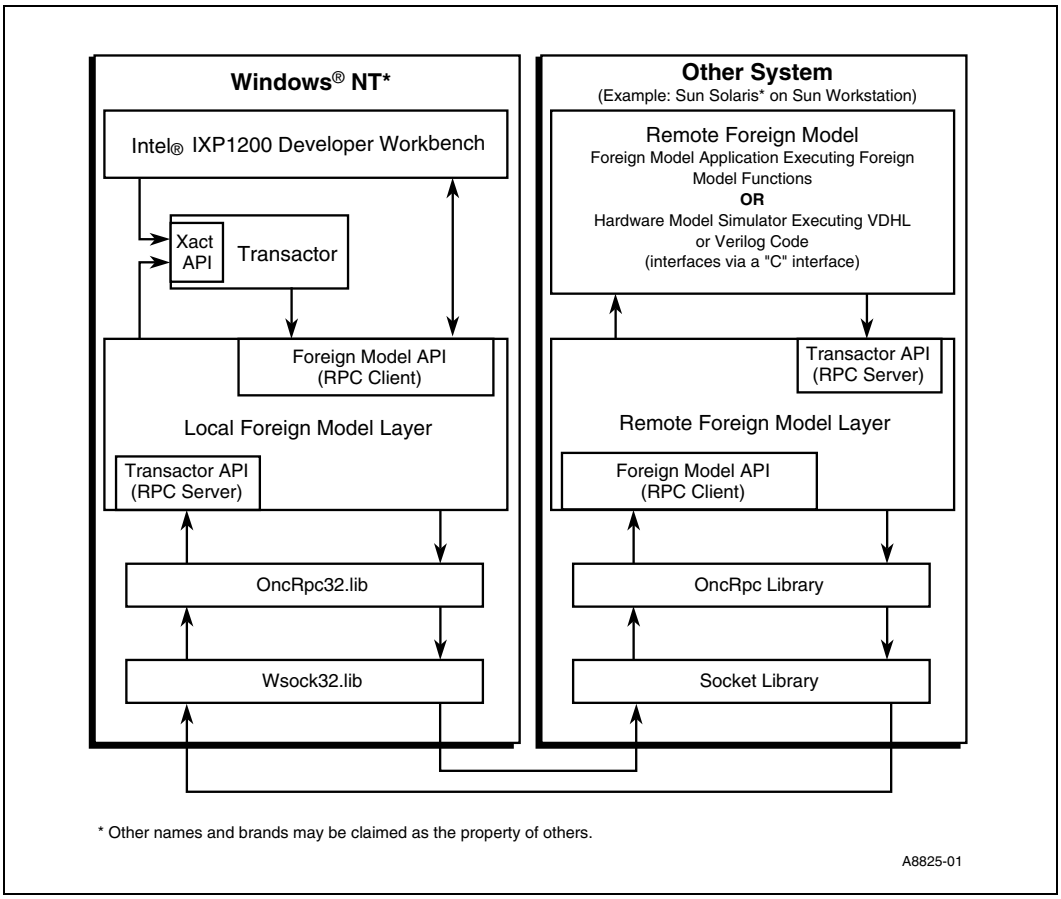
ONC RPC makes it appear as though a foreign model executable calls a procedure located in a Transactor program. ONC RPC was originally designed by Sun Microsystems and is in the public domain. Intel provides a Microsoft Windows NT version of ONC RPC (OncRpc32.lib) since the native support for RPC under Windows NT is different than ONC RPC.

The Workbench and Transactor are WindowsNT applications that use the OncRpc32.lib communications library to act as the communication layer between WindowsNT and UNIX, VxWorks, or other WindowsNT systems on the network. This communication library stands between user application and Wsock32. The Local and Remote Foreign Model Layers provide the RPC client and server functions between the Transactor and the remote foreign model. Intel

provides sample source code for the foreign model layers on both the WindowsNT and remote sides. These files need to be modified to provide the specific functionality for each implementation.

Figure 7-4 shows how the foreign model interface can be used to interface the IXP1200 Workbench to a Foreign Model Application or a hardware model written in hardware language such as VHDL or Verilog.

**Figure 7-4. Foreign Model Interface**




As the figure shows, the Transactor calls a local foreign model layer function instead of the actual foreign model. The foreign model layer function converts the input parameters as needed into a standard network data representation format, and then calls functions in the runtime RPC library, OncRpc32.lib, to send the request and its parameters to the remote foreign model.

The remote foreign model performs the following steps to call the actual foreign model functions:

1. The OncRPC library runtime library functions accepts the request and calls the remote foreign model layer.
2. The foreign model layer converts the parameters from the network transmission format to the format that the remote foreign function needs.
3. The remote model foreign layer function invokes the remote foreign function.

The remote foreign model functions interact with the Transactor through the remote foreign model layer DLL (RS_Xact_DLL.DLL). The Xact API for the remote Transactor is defined in the header file rs_xact.h. The Transactor layer invokes the actual Transactor function via the OncRPC mechanism.

## 7.3 Simulating StrongARM Core Applications

The Developer's Workbench can simulate a StrongARM Core application in conjunction with Microengine code. It supports the following methods for enabling the simulation:

- Invoke the StrongARM hardware simulator in the Workbench and run the core image
- Code the StrongARM Core application using Foreign Model techniques

The first method is accomplished by specifying the name of the StrongARM Core image in the Simulation Options dialog selected from the Debug menu of the Workbench. A StrongARM Core image is comprised of the operating system (e.g., VxWorks) and the core application. The Workbench provides a cycle-accurate simulation engine of the IXP1200 StrongARM Core under which the StrongARM Core image executes. This approach results in a precisely simulated interaction between the StrongARM Core and the Microengines. However, the cycle-accurate model used to simulate the StrongARM Core hardware may significantly increase the overall execution time of a simulation, especially when large core applications are involved. Consequently, this method should only be used when the core application contains limited functionality, such as performing the hardware boot sequence.

The second testing method uses the Transactor to provide a communication path between the StrongARM Core application and the Microengine code. While this technique does not result in a cycle-accurate model of the StrongARM Core application, it does provide a functionally accurate model of the interaction between the StrongARM Core and the Microengines while improving the overall simulation time. This section describes guidelines to code and build an application using Foreign Model interfaces that is targeted at both the StrongARM Core hardware and Transactor.

This section covers the following topics:

- Use Hardware Abstraction Layer (HAL) APIs
- Determine the IOSTYLE compiler directive
- Determine the coding considerations
- Accessing Foreign Model Functions from the C Interpreter
- Build the application to selected target format: Local Foreign Model or Remote Foreign Model

### 7.3.1 Using the Hardware Abstraction Layer (HAL)

The HAL API is documented in the *IXP1200 Network Processor Software Reference Manual*. The HAL API defines the function calls for accessing CSRs and functional unit memory such as SRAM, SDRAM and Scratchpad. To ensure portability of your StrongARM Core application between the hardware and the Transactor, the HAL must be used for memory access rather using direct memory-mapped reads and writes.

The HAL consists of set header files that define macros and function prototypes. These definitions provide mappings to the appropriate underlying target system - either the Hardware or the Workbench/Transactor. Compiler directives, in particular the IOSYTLE directive, control the underlying mapping.

## 7.3.2 Determining IOSTYLE

The IOSTYLE compiler directive controls how the HAL function calls and macros are compiled. All of the StrongARM Core application code must be built using the same IOSTYLE directive.

There are three possible IOSTYLE definitions:

**IOSTYLE=HARDWARE** Use this directive when building the core application to run on the StrongARM Core. HARDWARE directs the HAL to write to the memory-mapped addresses for functional unit memory and CSRs. If this directive is defined, you must also define the OS macro as either **OS=NT**, **OS=VXWORKS**, or **OS=UCOS**.

**IOSTYLE=AMBAIO** Use this directive when building the core application to run under the Transactor. AMBAIO supports access to all functional unit memory and CSRs. AMBAIO simulates access to functional units via the AMBA Bus which bypasses command queues used by the Microengine. This models the actual behavior of the StrongARM hardware.

**IOSTYLE=XACTIO** Use this directive when building the StrongARM Core application to run under the Transactor. Unlike AMBAIO, it does not allow access to all of the HAL functions. There is no access to CSRs or SRAM atomic functions when XACTIO is used,

## 7.3.3 Coding Considerations

Note the following when developing a StrongARM Core application that can be built as a Foreign Model:

- The Foreign Model entry points must be included in the application so that the Transactor can call them. Additionally, the function **GetForeignModelFunctions()** must be provided if creating the Foreign Model as a Windows NT DLL. Refer to Section 6.4.1 for a description of these entry functions. The compiler directive **ext_model_port** can be specified to automatically generate the external references to the Foreign Model entry points.
- Regardless of whether IOSTYLE is defined as XACTIO or AMBAIO, the function **ambaIoInit()** must be called by the StrongARM Core application if it uses the Microengine library (ueng.lib).
- The Transactor does not simulate IX Bus devices such as a MAC controller. If such modeling is required by the application, the device simulation must be provided separately, either in the form of a C code library or using the C Interpreter scripting tool in the Workbench. See section 6.2 for information on simulating an IX Bus device using Foreign Model techniques.
- When writing large amounts of functional unit memory, as when initializing all of SDRAM, the AMBAIO versions of the write() functions may take a significant amount of execution time. Alternatively, the functions XactFastRead() or XactFastWrite()perform block IO. Use these functions instead, but only in the Foreign Model version of the StrongARM Core application.
- The Transactor APIs (defined in xact.h) must only be used when running in simulation mode under control of the Transactor. Libraries built for any of the StrongARM Core operating systems do not support these calls.

### 7.3.4 Accessing Foreign Model Functions from the C Interpreter

The C Interpreter is described in Section 6.5. The Interpreter supports a subset of the C programming language and is used to create and run scripts on behalf of the Workbench and Transactor.

A StrongARM Core application using Foreign Model techniques can register function entry points, thus making them callable from a C script. To register an entry point in Simulation mode, use the XACT_register_console_function() call during initialization of the StrongARM Core application. This call is described in the *IXP1200 Network Processor Software Reference Manual*, Section 3.2.2.5.

Using a combination of scripts and Foreign Model code enables various configurations of the hardware and StrongARM Core application to be tested while not requiring a recompilation of the application code. Scripts can be written to initialize the state of the simulation, functional unit memory, and CSRs. They can also be written to simulate MAC devices to generate packet traffic on the IX Bus.

### 7.3.5 Building the Application

Once the design decisions are made, the following steps provide a guide to building the StrongARM Core application as a Foreign Model:

- Determine if the StrongARM Core application will run as a Remote Foreign Model or Local Foreign Model. If using the Local Foreign Model the application must be built as a Windows DLL. If using the Remote Foreign Model, the target operating system must support ONC RPC. See section 6.1 for more details on Local and Remote Foreign Models.
- The StrongARM Core application must include the header file **xact.h**. Additionally, use of the HAL API requires inclusion of the appropriate header files: **hal_sram.h**, **hal_sdram.h** and **hal_fbi.h**.
- The Foreign Model application must link a minimal set of libraries consisting of: **hal_1200.lib**, **xact_dll.lib**, **xactio.lib** and **utils.lib**. Other libraries are linked if used. For a description of the available libraries, see the *IXP1200 Network Processor Software Reference Manual*.

## 7.4 Simulating IX Bus Devices

Simulating of IX Bus devices involves the following:

- Getting states of IX Bus pins,
- Determining appropriate action based on the pin states, and
- Setting the appropriate pin states

The Transactor state names and a brief description of the states for various IX Bus pins is shown in Table 7-1. Section 3.3.4 (IX Bus Interface Pins) and Section 3.7 (IX Bus Decode Table) in the *IXP1200 Network Processor Datasheet* gives details of the usage of these pins. Timing diagrams showing the signal protocol can be found in Section 4.3.7.3 (IX Bus Protocol) and 4.3.7.4 (RDYBus). Other Transactor states are described in Appendix A, “Transactor States”.

In order to read or write data into Transactor states, a handle to the state must be first obtained using the Transactor API call **XACT_get_handle**(char *state_name, -1) Note that the second argument for this call is an array index. Since the pins are not arrayed states -1 should be used as the second argument. If multiple devices are attached to the bus, then each device should added as a bus client using the **XACT_add_bus_client**( char *bus_name, char *bus_client_name) API call before the handle is obtained. This ensures that bus errors will be detected and reported. Bus errors occur when multiple bus clients are simultaneously driving the bus, or when client reads invalid data because the bus was not driven. Note that internally Transactor treats each of the states listed in Table 7-1 as a separate "bus". Thus, each of these states require a separate call to add a bus client.

Once a handle is obtained for the state (or bus client) **XACT_get_state_value**(XACT_HANDLE state_handle, unsigned int *value)and **XACT_set_state_value**(XACT_HANDLE state_handle, unsigned int *value)can be used to read and write to the bus.

The device that is simulated needs to perform the following operations:

- Check which port is selected for driving ready flags and put out the appropriate ready flags on the Ready Bus.
- Check if flow control is being asserted and read the Ready Bus to obtain the ports for which flow control is asserted.
- Read port control outputs to determine which device is selected to drive (for receives by IXP1200) or sink (for transmits by IXP1200) and appropriately drive or sink the bus.

***Note:*** The set of pins that need to be driven or read from depends on the IX Bus mode selected. Please refer to the *IXP1200 Network Processor Datasheet* for details.

The foreign model is called for every Transactor simulation clock cycle. The simulation clock cycle can be configured to either IX Bus clock or core clock. This can be selected on the Developer's Workbench under Debug Simulator → Options → Configuration tab → FIFO Bus Clock or Core Clock check box. The foreign model must drive/read the IX Bus pins in synchronization with the IX Bus clock. If the simulation is configured for IX Bus clock, no special synchronization is needed and reads/writes to the bus can performed in the post_sim() or pre-sim() calls in Foreign Model API (Section 6.5.1).

If the simulation is configured for the Core clock, the foreign model should synchronize with the IX Bus clock using one of the following:

- Use **XACT_Define_Callback_State_Transition**() with a handle to the sim.fbus_clk_cycle to register a function that is called for each IX Bus clock cycle. Read and writes to the bus are performed in this function.
- Use **XACT_clock_cycle_with_remainder**() or **XACT_clock_cycle**() in pre_sim() or post_sim() Foreign Model functions to determine if the IX Bus clock has changed. Reads and writes to bus are performed in the post_sim() or pre-sim() calls in Foreign Model API.

**Table 7-1. Transactor Names for IX Bus Interface Pins**

| Pin Name | Transactor Name | Description |
|---|---|---|
| FDAT[31:0] | fdat_lo | Lower 32 bits of the IX data bus |
| FDAT[63:32] | fdat_hi | Upper 32 bits of the IX data bus |
| FBE#[3:0] | fbe_lo_l | Bi-directional byte enable bits <0:3> |
| FBE#[7:4] | fbe_hi_l | Bi-directional byte enable bits <4:7> |

**Table 7-1. Transactor Names for IX Bus Interface Pins (Continued)**

| Pin Name | Transactor Name | Description |
|---|---|---|
| RXFAIL | rx_fail | Receive Packet Failure |
| EOP | eop | End of Packet Indication |
| SOP | sop | Start of Packet Indication |
| EOP32_XMIT | eop32_xmit | Transmit End of Packet Indication |
| SOP32_XMIT | sop32_xmit | Transmit Start of Packet Indication |
| FAST_RX1 | fast_rx1 | Ready input from Fast Port 1 |
| FAST_RX2 | fast_rx2 | Ready input from Fast Port 2 |
| PORTCTL#[3:0] | portctl_1 | Port Control outputs |
| FPS[2:0] | fps | MAC Port Select outputs |
| RDYBUS[7:0] | rdybus | Ready Bus I/O |
| RDYCTL[3:0] | rdyctl_1 | Ready control bits <0:3> |
| RDYCTL[4] | rdyctl_4 | Ready control bit <4> |
| TK_IN | token_in | Token in |
| TK_OUT | token_out | Token out |
| GPIO[3:0] | gpio | General Purpose I/O |

## 7.5 Sample Code

The sections that follow contain some sample code.

### 7.5.1 Creating A Foreign Model DLL Used with the Developer Workbench

This procedure describes how to create a foreign model layer (DLL) to be used in the foreign model DLL configuration.

- The foreign model DLL must provide the exported function GetForeignModelFunctions() in addition to the foreign_model_initialize(), foreign_model_pre_sim(), foreign_model_post_sim(), foreign_model_reset(), and foreign_model_exit() functions.
- The file xact_dll.h must be included in foreign model DLL source files, if needed, and it must be linked against xact_dll.lib.

#### 7.5.1.1 DLL Sample Code

```
// You must include xact_dll.h, NOT xact.h in order to link correctly against
// the DLL version of the Transactor.
//
#include "xact_dll.h"

int foreign_model_initialize()
{
    …your initialization code…
    return(1);
}

int foreign_model_pre_sim()
{
    … your pre simulation event code…
    return(1);
}

int foreign_model_post_sim()
{
    … your post simulation event code…
    return(1);
}

int foreign_model_reset()
{
    … your reset code…
    return(1);
}

int foreign_model_exit()
{
    … your exit code…
    return(1);
}
```

```
/*-----------------------------------------------------------------
    GetForeignModelFunctions

    This function is exported as the sole entry point into the DLL version
    of this package. The Developer Workbench calls it in order to get the
    foreign model entry points for the Transactor. The Workbench then
    registers these entry points with the Transactor.

     returns:
        uses:
    modifies:
*/
#if defined(__cplusplus)
extern "C"      // the exported function must be of generic C type (not CPP),
                // so that its name doesn't get adorned.
#endif
__declspec(dllexport) void __cdecl
GetForeignModelFunctions(
    int (**ForeignModelInitialize)(),
    int (**ForeignModelPreSim)(),
    int (**ForeignModelPostSim)(),
    int (**ForeignModelReset(),
    int (**ForeignModelExit)())
{
    *ForeignModelInitialize = foreign_model_initialize;
    *ForeignModelPreSim = foreign_model_pre_sim;
    *ForeignModelPostSim = foreign_model_post_sim;
    *ForeignModelExit = foreign_model_exit;
    *ForeignModelReset = foreign_model_reset;
}}
```

## 7.5.2 Creating A Remote Foreign Model - Local Foreign Model Layer

The procedure that follows describes how to create a Local and Remote Foreign Model layer (DLL) to be used in the Remote Foreign Model Executable configuration. The procedure is the same used to create a local foreign model DLL in the Foreign Model Dynamic-Link Library (DLL) configuration with the addition of the following steps.

- OncRpb32.lib and Wsock32.lib must also be linked.
- Rs_xact.h must be included in the source files.
- OncRpc32Init() must be called before making any OncRpc calls, and call OncRpc32Cleanup() before exit.

### 7.5.2.1 Sample Code

```
/*
foreign_model_initialize
This sample function calls the OncRpc32Init to initialize the network layers,
register a local foreign model function that performs the remote network
initialization. It invokes the local rs_foreign_model_initialize() stub function
```

```
which then calls clnt_call(), a OncRpc32 library function to send a request to the
remote foreign model application to execute the actual foreign_model_initialize()
function.
 */
int foreign_model_initialize()
{
    OncRpc32Init();

   if (!XACT_register_console_function("RemoteInit", RemoteInit, 1, 1) ){
            fprintf(stderr,"XACT_register_console_function() failed\n");
           return(0);
    }
   rs_foreign_model_initialize();
    return(1);
}

/*-------------------------------------------------------------------
foreign_model_pre_sim
This function calls a local rs_foreign_model_pre_sim stub function which then calls
clnt_call(), a OncRpc32 library function to send a request to the remote foreign
model application to execute the actual foreign_model_pre_sim() function.
*/
int foreign_model_pre_sim()
{
   rs_foreign_model_pre_sim();
   return(1);
}

/*-------------------------------------------------------------------
foreign_model_post_sim
This function calls a local rs_foreign_model_post_sim stub function which then
calls clnt_call(), an OncRpc32 library function to send a request to the remote
foreign model application to execute the actual foreign_model_post_sim() function.
*/
int foreign_model_post_sim()
{
   rs_foreign_model_post_sim();
   return(1);
}

/*-------------------------------------------------------------------
foreign_model_post_sim
This function calls a local rs_foreign_model_reset stub function which then calls
clnt_call(), an OncRpc32 library function to send a request to the remote foreign
model application to execute the actual foreign_model_reset() function.
*/
int foreign_model_reset()
{
   rs_foreign_model_reset();
   return(1);
}

/*
foreign_model_exit_sim
This function calls a local rs_foreign_model_exit stub function which then calls
clnt_call(), a OncRpc32 library function to send a request to the remote foreign
model application to execute the actual foreign_model_exit_sim() function.
```

```
*/

int foreign_model_exit()
{
  rs_foreign_model_exit();
    OncRpc32Cleanup();
  return(1);
}
```

## 7.5.3 Creating a Remote Foreign Model - Remote Foreign Model Layer

The procedure that follows describes how to create a Remote Foreign Model Layer (DLL) to be used in the Remote Foreign Model Executable configuration. If you choose to create a server thread that will get a transport handle used for receiving and replying to remote foreign model calls, the foreign model must be linked against rs_xact_dll.lib, OncRpc, and socket libraries and rs_xact.h must be included in the source files.

### 7.5.3.1 Sample Code

```
/*
 * The remote version of  "rs_foreign_model_initialize"
 * you must call ConnectionToXact () to establish the connection with Transactor
before you make any  *Transactor function calls.
 */


int *
rs_foreign_model_initialize()
{

    static int result =1; /* must be static! */
  RS_XACT_HANDLE hSimTime;

/*
  *
 * *XactServer is the hostname where the Workbench is running on
 */
  if(ConnectionToXact(*XactServer) != RS_SUCCESS){
        printf("open_connection failed\n");
        result=0;
    }
   /*
    *  For example, you are making a Transactor call
    */
    hSimTime = RS_XACT_get_handle("sim.time", -1);
   if(hSimTime == RS_INVALID_XACT_HANDLE)
        printf("ErrorError - RS_XACT_get_handle failed\n");

… your initialization code

   return &result;
}
```

```
/*
 * The remote version of "rs_foreign_model_pre_sim"
 */

int *
rs_foreign_model_pre_sim()
{
    static int result=1; /* must be static! */
   … your pre simulation event code
    return (&result);

}

/*
 * The remote version of "rs_foreign_model_post_sim"
 */

int *
rs_foreign_model_post_sim()
{
    static int result=1; /* must be static! */

… your pre simulation event code

    return (&result);

int *
rs_foreign_model_reset()
{
    static int result=1; /* must be static! */

… your reset code

    return (&result);

}


/*
 * The remote version of "rs_foreign_model_exit"
 */

int *
rs_foreign_model_exit()
{
    static int result=1; /* must be static! */
  … your exit code
    return (&result);
}
```

# Transactor States A

## A.1 About States

The IXP1200 Transactor contains internal states that define the overall state of the model. The examine command with the asterisk wildcard character (**examine ***) can be used to view all the model states. This section describes only the most commonly used model states. States can be grouped into three categories:

- Hardware states. Affect SRAM, SDRAM, registers, FIFOs, etc.
- Simulator states. Affect simulator functions such as debugging and error reporting, starting and stopping the Transactor, etc.
- Statistics states. Affect the reporting of SRAM, SDRAM, FBus, and Microengine performance statistics.

States can be read and written by the user using the examine and deposit Transactor commands. Any simple model state (i.e., a signal, register, or array (not a FIFO)) is also treated as a global int when it appears in a C statement or expression. States can be preserved and recalled using the save and restore Transactor commands.

Table A-1 is a quick reference to the IXP1200 Transactor states described in this chapter:

**Table A-1. IXP1200 Transactor States**

| State Name | State Format | Page |
|---|---|---|
| **Hardware States** | | |
| Microengine GPR or Transfer Register | {chipname}.fx.reg.register_name_thd | A-5 |
| Microengine Control Store | {chipname}.fx.ctl.ustore{[addr]<msb:lsb>} | A-6 |
| Microengine Condition Codes | {chipname}.fx.ctl.condition_code | A-7 |
| Microengine Microstore Data Register | {chipname}.fx.ctl.csr_ustore_rd_data<br>{chipname}.fx.ctl.csr_ustore_wr_data | A-8 |
| Microengine Microstore Address Register | {chipname}.fx.ctl.csr_ustore_rd_address<br>{chipname}.fx.ctl.csr_ustore_wr_address | A-8 |
| Microengine Thread Program Counter Register | {chipname}.fx.ctl.pc_ctx_x | A-9 |
| Enabled Thread Wake-Up Signals | {chipname}.fx.ctl.thread_trigger_mask_x | A-10 |
| Active Thread Wake-Up Signals | {chipname}.fx.ctl.enabled_signals_x | A-11 |
| Transmit and Receive FIFOs | {chipname}.fbi.rfifo{[addr]<msb:lsb>}<br>{chipname}.fbi.tfifo{[addr]<msb:lsb>} | A-12 |
| FBI CSRs | {chipname}.fbi.csr{[addr]<msb:lsb>} | A-13 |
| FBI Scratchpad Memory | {chipname}.fbi.scratch_pad{[addr]<msb:lsb>} | A-15 |
| SRAM | {chipname}.sram{[addr]<msb:lsb>} | A-16 |
| SRAM Push/Pop Registers | {chipname}.sc.push_pop_regs{[addr]<msb:lsb>} | A-17 |
| SDRAM | {chipname}.sdram{[addr]<msb:lsb>} | A-18 |

### Table A-1. IXP1200 Transactor States (Continued)

| State Name | State Format | Page |
|---|---|---|
| **Simulation States** | | |
| Specify Microengine(s) for GOTO | {chipname}.art.fbox_mask | A-19 |
| Specify Thread(s) for GOTO | {chipname}.fx.art.ctx_mask<br>{chipname}.art.ctx_mask | A-20 |
| Report Core Clock Cycles | sim.core_clk_cycle | A-20 |
| Core Clock Frequency | sim.core_clk_freq | A-21 |
| Core Clock Period | sim.core_clk_period | A-21 |
| Report FBus Clock Cycles | sim.fbus_clk_cycle | A-21 |
| FBus Clock Frequency | sim.fbus_clk_freq | A-22 |
| FBus Clock Period | sim.fbus_clk_period | A-22 |
| Report PCI Clock Cycles | sim.pci_clk_cycle | A-22 |
| PCI Clock Frequency | sim.pci_clk_freq | A-23 |
| PCI Clock Period | sim.pci_clk_period | A-23 |
| Execution Order | sim.post_sim_exec_order | A-24 |
| Report Error Count | sim.error_count | A-24 |
| Error Handle Mode | sim.error_handle_mode | A-25 |
| Stop Simulation on P1 or P3 | sim.goto_and_ubreak_key_on_p1 | A-25 |
| Halt Simulation | sim.halt | A-26 |
| Suppress Comments | sim.prune_prefix_comments | A-26 |
| Suppress Debug Information | sim.silent | A-27 |
| Report Simulation Time | sim.time | A-27 |
| Specify Debug Verbosity Level | {chipname}.unit.art.debug | A-28 |
| Specify Reported Execution Stages | {chipname}.fx.art.debug_px | A-29 |
| **Statistics States** | | |
| Compute Cycles | {chipname}.fx.stat.compute_cycles | A-32 |
| SDRAM Read Latency | {chipname}.fx.stat.sdram_read_latency | A-33 |
| SRAM Read Latency | {chipname}.fx.stat.sram_read_latency | A-33 |
| SRAM Read Lock Latency | {chipname}.fx.stat.sram_read_lk_latency | A-34 |
| Thread PC During Idle Cycle | {chipname}.fx.stat.threadx_pc_during _idle_cycles | A-34 |
| Thread Latency | {chipname}.fx.stat.thread_latency | A-35 |
| Time Slice | {chipname}.fx.stat.time_slice_threadx | A-36 |
| Time Slice Thread Idle | {chipname}.fx.stat.time_slice_thread_idle | A-36 |
| SDRAM Bandwidth | {chipname}.dc.stat.bandwidth | A-37 |
| SDRAM Even Bank Queue Fullness | {chipname}.dc.stat.even_bank_queue_fullness | A-37 |
| SDRAM Odd Bank Queue Fullness | {chipname}.dc.stat.odd_bank_queue_fullness | A-38 |
| SDRAM Order Queue Fullness | {chipname}.dc.stat.order_queue_fullness | A-38 |
| SDRAM Priority Queue Fullness | {chipname}.dc.stat.priority_queue_fullness | A-39 |

**Table A-1. IXP1200 Transactor States (Continued)**

| State Name | State Format | Page |
|---|---|---|
| SRAM Bandwidth | {chipname}.sc.stat.bandwidth | A-39 |
| SRAM Order Queue Fullness | {chipname}.sc.stat.order_queue_fullness | A-40 |
| SRAM Priority Queue Fullness | {chipname}.sc.stat.priority_queue_fullness | A-40 |
| SRAM Read Lock Queue Fullness | {chipname}.sc.stat.rd_lk_queue_fullness | A-41 |
| SRAM Read Queue Fullness | {chipname}.sc.stat.read_queue_fullness | A-41 |
| SRAM State Machine Distribution | {chipname}.sc.stat.state_machine_dist | A-42 |
| Command Bus Bandwidth | {chipname}sc.stat.cmd_bus_bandwidth | A-42 |
| SBUS Pull Bus Bandwidth | {chipname}.stat.sbus_pull_bandwidth | A-43 |
| SBUS Push Bus Bandwidth | {chipname}.stat.sbus_push_bandwidth | A-44 |
| SDRAM Pull Bus Bandwidth | {chipname}.stat.sdram_pull_bandwidth | A-45 |
| SDRAM Push Bus Bandwidth | {chipname}.stat.sdram_push_bandwidth | A-46 |

## A.1.1 State Definition Format

This section lists the most commonly used Transactor states in alphabetical order and describes each one. The state definitions in the sections that follow contain:

- The state name (e.g., {chipname}.fx.reg.register_name_thd).
- A description of the state.
- A description of the parameters, if any, associated with the state.
- One or more examples illustrating the use of the state.

Optional input strings are delimited by bracket characters ({}) in these descriptions.

**Bolding** indicates a predefined, required part of the state definition.

## A.1.2 Display Formats for Statistics States

Statistics states are used with the **examine** command to obtain microcode performance information. Depending on the state, statistics are reported in one of two formats.

**Format 1:**

```
samples = 12; min = 17; ave = 20; max = 36; data units = cycles
   sample distrib |  4|  5|  1|  1|   1|
   percent        |33%|42%| 8%| 8%|  8%|
   cumul percent  |33%|75%|83%|92%|100%|
   cycles         | 17| 18| 24| 30|  36|
```

The first line of text indicates the following:
samples = The total number of samples taken for these statistics.
min = The minimum number (of cycles, in this case).
ave = The average number (of cycles, in this case).
max = The maximum number (of cycles, in this case).
data units = The units in which the data (min, ave, and max) is expressed.

The table shows the distribution of samples by category. In this case, samples are categorized by cycle count. The 12 samples taken for these statistics fall into five categories of cycle counts: 17, 18, 24, 30, and 36. Four samples had a cycle count of 17, five had a cycle count of 18, and so on. The table contains the following information:

sample distrib = The number of samples in each category.
percent = The percentage of samples in each category.
cum percent = The cumulative percentage of samples by category.
cycles = Category. In this case, samples are categorized by cycle count.

**Format 2:**

```
samples = 24; min = 0; ave = 4; max = 9; data units = cycles
ctx 0 non-aborted cycles:  6     (25%) ************
ctx 0 aborted cycles:      1     (4%)  **
ctx 0 stalled cycles:      0     (0%)
ctx 1 non-aborted cycles:  6     (25%) ************
ctx 1 aborted cycles:      0     (0%)
ctx 1 stalled cycles:      0     (0%)
ctx 2 non-aborted cycles:  6     (25%) ************
ctx 2 aborted cycles:      0     (0%)
ctx 2 stalled cycles:      0     (0%)
ctx 3 non-aborted cycles:  5     (21%) **********
ctx 3 aborted cycles:      0     (0%)
ctx 3 stalled cycles:      0     (0%)
no ctx active cycles:      0     (0%)
```

Format 2 categorizes samples (in this case) according to context and cycle type. The information in the first line is identical to that of Format 1. The table, however, features category in column 1, the number of samples in each category in column 2, the percentage of samples in each category in column 3, and a graphical representation (horizontal bar chart) of the distribution of samples in column 4.

# A.2 Hardware States

## A.2.1 Microengine GPR or Transfer Register

Used with the deposit and examine commands.

**Format:** `{chipname}.fx.reg.register_name_thd`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fx`** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.

**`reg`** Constant that specifies this is a register state.

**`register_name_thd`**
The name of the register followed by the thread number. The thread number, can be 0, 1, 2, or 3. Can appear in one of three forms, where x is the thread number:

- For GPRs: regname_x (e.g., reg0_1 for the GPR named reg0 associated with thread 1)
- For SRAM transfer registers: $regname_rd_x or $regname_wr_x (e.g., $reg0_rd_1 for the SRAM read transfer register named reg0 associated with thread 1).
- For SDRAM transfer registers: $$regname_rd_x or $$regname_wr_x (e.g., $$reg0_wr_1 for the SDRAM write transfer register named reg0 associated with thread 1).

**Example 1:**

```
> deposit f0.reg.$val_rd_0 = 0xffff
```

Deposit some data into an SRAM transfer register named $val_rd. Chip has a null (no) name. We want Microengine 0, thread 0.

To examine the data:

```
> examine f0.reg.$val_rd_0
```

Example 2:

```
> examine chip1.f0.temp_0
```

Examine a GPR named temp in a chip named chip1. Examine Microengine 0, thread 0.

## A.2.2 Microengine Control Store

Used with the **examine** command. The Microengine control store is an array of 128 longwords.

**Format:** {chipname}.fx.**ctl**.ustore{[addr]<msb:lsb>}

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**ctl** Constant that specifies this is a model control signal.

**ustore{[addr]<msb:lsb>}**
If [addr] is used, it specifies an offset into the program store. Valid values are 0 to 1023. Single locations are specified by a single value (e.g., [0]). A range of locations is specified by using maximum and minimum values separated by a colon (e.g., [1023:0]). Bits within a longword can be specified by <msb:lsb>. If [addr] is not used, information on the number of valid words in the control store and the file from which the control store was loaded are displayed.

**Example 1:**
```
> examine f0.ctl.ustore
```
Examine the control store without specifying an [addr]. Example result:
```
f0.ctl.ustore:  ustore of 1024 32-bit words;  23 words
currently valid.
Ucode loaded from: "llist.list"; contains 1 pages
```

**Example 2:**
```
> ex f0.ctl.ustore[0]
```
Examine a single location in the control store.

**Example 3:**
```
> ex f0.ctl.ustore[3:0]
```
Examine locations 3 through 0 in the control store.

### A.2.3 Microengine Condition Codes

Used with the **examine** and **deposit** commands. There are three single bit condition codes: Carry, Negative, and Zero. A condition code is set to 1 when its corresponding condition occurs.

**Format:** `{chipname}.fx.ctl.condition_code`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fx`** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.

**`ctl`** Constant that specifies this is a model control signal.

**`condition_code`**

Condition code name. Valid names are:

```
cc_carryCarry
cc_negNegative
cc_zeroZero
```

**Example 1:**

```
> examine f0.ctl.cc_neg
```

Examine the Negative condition code. Example result indicating 0 as the Negative condition code:

```
f0.ctl.cc_neg<0>:  0x0
```

**Example 2:**

```
> examine f0.ctl.cc_neg
```

Examine the Carry condition code. Example result indicating 1 as the Carry condition code:

```
f0.ctl.cc_carry<0>:  0x1
```

## A.2.4 Microengine Microstore Data Registers

Used with the **examine** and **deposit** commands. Each Microengine has a 32-bit microstore data read register and a 32-bit microstore data write register, each of which has its own state definition.

**Formats:**
{chipname}.fx.**ctl**.csr_ustore_rd_data
{chipname}.fx.**ctl**.csr_ustore_wr_data

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.

**ctl** Constant that specifies this is a model control signal.

**csr_ustore_rd_data**
Specifies the read microstore data register.

**csr_ustore_wr_data**
Specifies the write microstore data register.

**Example:**
```
> ex f1.ctl.csr_ustore_rd_data
```
Examine the microstore data read register of Microengine 1.

## A.2.5 Microengine Microstore Address Registers

Used with the examine and deposit commands. Each Microengine has a 10-bit microstore address read register and a microstore address write register, each of which has its own state definition.

**Formats:**
{chipname}.fx.**ctl**.csr_ustore_rd_address
{chipname}.fx.**ctl**.csr_ustore_wr_address

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.

**ctl** Constant that specifies this is a model control signal.

**csr_ustore_rd_address**
Specifies the read microstore address register.

**csr_ustore_wr_address**
Specifies the write microstore address register.

**Example:**
```
> ex f1.ctl.csr_ustore_rd_address
```
Examine the microstore address read register of Microengine 1.

## A.2.6 Microengine Thread Program Counter Register

Used with the examine and deposit commands. Each Microengine has four 10-bit thread program counter registers.

**Formats:** `{chipname}.fx.ctl.pc_ctx_x`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fx`** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.

**`ctl`** Constant that specifies this is a model control signal.

**`pc_ctx_x`** Thread program counter register. Valid names are:

- `pc_ctx_0` Program counter register for thread 0.
- `pc_ctx_1` Program counter register for thread 1.
- `pc_ctx_2` Program counter register for thread 2.
- `pc_ctx_3` Program counter register for thread 3.

**Example:**

```
> ex f0.ctl.pc_ctx_1
```

Examine the program counter register for thread 1 of Microengine 0.

## A.2.7 Enabled Thread Wake Up Signals

Used with the examine and deposit commands. This state involves an 11-bit mask, each bit of which corresponds to a signal from one of the following sources:

**BitSource**
0self (current Microengine)
1SRAM
2SDRAM
3Interthread 1
4Interthread 2
5Receive request available
6FBI 1
7FBI 2
8Sequence number 1
9Sequence number 2
10PCI

When a bit in the bit mask is set, that source is allowed to wake up a thread. When a bit is cleared, that source can not wake up a thread.

**Formats:** `{chipname}.fx.ctl.thread_trigger_mask_x`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fx`** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.

**`ctl`** Constant that specifies this is a model control signal.

**`thread_trigger_mask_x`**
Each thread can have its own mask. Valid names are:

- `thread_trigger_mask_0` Mask for thread 0.
- `thread_trigger_mask_1` Mask for thread 1.
- `thread_trigger_mask_2` Mask for thread 2.
- `thread_trigger_mask_3` Mask for thread 3.

**Example 1:**
```
> dep/multi f*.ctl.thread_trigger_mask* = 0x111
```
Enable all threads in all Microengines to wake up as a result of a wake up signal from any source.

**Example 2:**
```
> dep/multi f*.ctl.thread_trigger_mask_0 = 0x0
```
Disable thread 0 in all Microengines from being woken up by any source.

## A.2.8 Active Thread Wake Up Signals

Used with the examine and deposit commands. This state involves an 11-bit mask, each bit of which corresponds to a signal from one of the following sources:

**BitSource**
0Self (i.e., the current Microengine)
1SRAM
2SDRAM
3Interthread 1
4Interthread 2
5Receive request available
6FBI 1
7FBI 2
8Sequence number 1
9Sequence number 2
10PCI

When a bit in the mask is set, that source is currently waiting to wake up a thread. When a bit is cleared, that source is not currently waiting to wake up a thread.

**Formats:** `{chipname}.fx.ctl.enabled_signals_x`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fx`** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.

**`ctl`** Constant that specifies this is a model control signal.

**`enabled_signals_x`** Each thread can have its own mask. Valid names are:

- `enabled_signals_0` Mask for thread 0.
- `enabled_signals_1` Mask for thread 1.
- `enabled_signals_2` Mask for thread 2.
- `enabled_signals_3` Mask for thread 3.

**Example:**
```
> ex f*.ctl.enabled_signals*
```
Use wildcards to check what sources are currently waiting to wake up any thread in any Microengine.

## A.2.9 Transmit and Receive FIFOs

Used with the deposit and examine commands. Refers to an array of 128 quadwords in the FBI unit's transmit and receive FIFOs.

**Format:**

```
{chipname}.fbi.rfifo{[addr]<msb:lsb>}
{chipname}.fbi.tfifo{[addr]<msb:lsb>}
```

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fbi** Constant that specifies the FIFO bus interface unit.

**rfifo{[addr]<msb:lsb>}**
[addr] specifies a quadword offset into the receive FIFO array. Valid [addr] values are 0 to 127. Note that the beginning of one of the sixteen 64-byte elements in the array occurs every eight quadwords. Bits within a quadword can be specified by <msb:lsb>. Deposits can only be performed for values of 32 bits or less. Examines can occur on 0 to 64 bit values. If [addr] is omitted, the state refers to the FBI receive FIFO as a whole.

**tfifo{[addr]<msb:lsb>}**
[addr] specifies a quadword offset into the transmit FIFO array. Valid [addr] values are 0 to 127. Note that the beginning of one of the sixteen 64-byte elements in the array occurs every eight quadwords. Bits within a quadword can be specified by <msb:lsb>. Deposits can only be performed for values of 32 bits or less. Examines can occur on 0 to 64 bit values. If [addr] is omitted, the state refers to the FBI transmit FIFO as a whole.

**Example 1:**

```
> examine fbi.rfifo[0]<63:0>
```

Examine the .... Example result:

```
array:fbi.rfifo[0]<63:0>:  00000000 00000000
```

**Example 2:**

```
> dep fbi.rfifo[0]<63:32> = 0xffffffff
```

Deposit the .... Example result:

```
array:fbi.rfifo[0]<63:0>:  FFFFFFFF 00000000
```

## A.2.10 FBI CSRs

Used with the deposit and examine commands. Refers to a 32-bit FBI CSR.

**Format:** `{chipname}.fbi.csr_name{<msb:lsb>}`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fbi`** Constant that specifies the FIFO bus interface unit.

**`csr_name{<msb:lsb>}`**
Specifies the CSR name, as summarized in Table A-2.

**Table A-2. FBI CSRs**

| Register Name | csr_name |
|---|---|
| IREG | ireg |
| SOP_SEQ1 | sop_seq1 |
| SOP_SEQ2 | sop_seq2 |
| ENQUEUE_SEQ1 | enqueue_seq1 |
| ENQUEUE_SEQ2 | enqueue_seq2 |
| THREAD_DONE_REG0 | thread_done1 |
| THREAD_DONE_REG1 | thread_done2 |
| THREAD_DONE_INCR1 | thread_done1 |
| THREAD_DONE_INCR2 | thread_done2 |
| RCV_RDY_CNT | rec_ready_count |
| RCV_RDY_HI | rec_ready_bits_hi |
| RCV_RDY_LO | rec_ready_bits_lo |
| RCV_RDY_CTL | rec_ready_control |
| RCV_REQ | rec_req |
| RCV_CNTL | rec_ctl |
| FLOWCTL_MASK | fbus.flowctl_mask |
| RDYBUS_TEMPLATE_CTL | fbus.rdybus_template_ctl |
| RDYBUS_TEMPLATE_PROG_1 | fbus.rdybus_template_ctl1 |
| RDYBUS_TEMPLATE_PROG_2 | fbus.rdybus_template_ctl2 |
| RDYBUS_TEMPLATE_PROG_3 | fbus.rdybus_template_ctl3 |
| RDYBUS_SYNC_COUNT_DEFAULT | fbus.rdy_synch_count_default |
| SELF_DESTRUCT | self_destruct |
| CYCLE_CNT_LO | cycle_count_lsw |
| CYCLE_CNT_HI | cycle_count_msw |
| HASH_MULTIPLIER_64_HI | hash_multiplier_64_hi |
| HASH_MULTIPLIER_64_LO | hash_multiplier_64_lo |
| HASH_MULTIPLIER_48_HI | hash_multiplier_48_hi |
| HASH_MULTIPLIER_48_LO | hash_multiplier_48_lo |

**Table A-2. FBI CSRs (Continued)**

| Register Name | csr_name |
|---|---|
| SEND_CMD | fbus.sendq |
| GET_CMD | getq |
| XMIT_RDY_LO | xmit_ready_bits_lo |
| XMIT_RDY_HI | xmit_ready_bits_hi |
| XMIT_RDY_CTL | fbus.xmit_ctl |
| XMIT_VALIDATE | xmit_valid_bits |
| XMIT_PTR | fbus.frozen_xmit_valid_outptr |
| FP_READY_WAIT | fbus.fp1_wait_count_csr<3:0><br>fbus.fp2_wait_count_csr<3:0> |
| REC_FAST_PORT_CTL | fast_port1_body_thread<4:0><br>fast_port2_body_thread<4:0><br>fast_port1_header_thread<4:0><br>fast_port2_header_thread<4:0> |

Bits within a CSR can be specified by <msb:lsb>. If [addr] is omitted, the state refers to the FBI CSR as a whole.

**Example 1:**

```
> dep fbi.csr[0] = 0xf1f1f1f1
```

Deposit a value into the THREAD_DONE_REG0 FBI CSR. Example result:

```
fbi.csr[0]<31:0>:  32
```

**Example 2:**

```
> ex fbi.csr[0]
```

Examine the THREAD_DONE_REG0 FBI CSR. Example result:

```
fbi.csr[0]<31:0>:  F1F1F1F1
```

## A.2.11 FBI Scratchpad Memory

Used with the deposit and examine commands. Refers to the FBI scratchpad memory, which is an array of 128 quadwords.

**Format:** `{chipname}.fbi.scratch_pad{[addr]<msb:lsb>}`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fbi`** Constant that specifies the FIFO bus interface unit.

**`scratch_pad{[addr]<msb:lsb>}`**
[addr] specifies an offset into FBI scratchpad. Valid [addr] values are 0 to 1023. Single locations are specified by a single value (e.g., [0]). A range of locations is specified by using maximum and minimum values separated by a colon (e.g., [1023:0]). Bits within a quadword can be specified by <msb:lsb>. Deposits can only be performed for values of 32 bits or less. Examines can occur on 0 to 64 bit values. If [addr] is omitted, the state refers to the FBI CSR as a whole.

**Example 1:**
```
> dep fbi.scratch_pad[0] = 0xffffffff
```
Deposit 0xffffffff into FBI scratchpad memory. Example result:
```
fbi.scratch_pad[0]<31:0>:  32
```

**Example 2:**
```
> ex fbi.scratch_pad[0]
```
Examine the same location. Example result:
```
fbi.scratch_pad[0]<31:0>:  FFFFFFFF
```

## A.2.12 SRAM

Used with the **deposit** and **examine** commands. Refers to the SRAM array of longwords, the size of which is set when you initialize memory with the mem_init Transactor command.

**Format:** `{chipname}.sram{[addr]<msb:lsb>}`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`sram{[addr]<msb:lsb>}`**
[addr] specifies a longword offset into SRAM. Valid [addr] values are 0 to the amount in bytes specified in the mem_init Transactor command divided by 4 (since there are 4 bytes in one longword). Single locations are specified by a single value (e.g., [0]). A range of locations is specified by using maximum and minimum values separated by a colon (e.g., [1023:0]). Bits within an SRAM longword can be specified by <msb:lsb>.If [addr] is omitted, the state refers to SRAM as a whole.

**Example 1:**
```
> chip
AA-1200 chip, "", was instantiated
> mem_init sram 1m
> dep sram[0] = 0xffffffff
memory:sram[0]<31:0>   FFFFFFFF (-1)
```
Instantiate a chip, use the mem_init command to set the size of SRAM, and deposit a value. The lines preceded by a prompt character (>) are the ones you type in.

**Example 2:**
```
> ex sram[0]
memory:sram[0]<31:0>   FFFFFFFF (-1)
```
Examine the result of Example 1.

**Example 3:**
```
> ex sram[0:4]
```
After changing the contents of SRAM, examine the first five longwords. Example result:
```
memory:sram[0]<31:0>:  00123456 (1193046)
memory:sram[1]<31:0>:  00123456 (1193046)
memory:sram[2]<31:0>:  0000000A (10)
memory:sram[3]<31:0>:  00000005 (5)
memory:sram[4]<31:0>:  00000014 (20)
```

## A.2.13 SRAM Push/Pop Registers

Used with the **deposit** and **examine** commands. Refers to any or all of eight 23-bit registers, each of which holds a pointer to the first element in a linked list in SRAM.

**Format:** `{chipname}.sc.push_pop_regs{[addr]<msb:lsb>}`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`sc`** Constant that specifies SRAM.

**`push_pop_regs{[addr]<msb:lsb>}`**
[addr] specifies one of eight push/pop registers. Valid [addr] values are 0 to 7. Single registers are specified by a single value (e.g., [0]). A range of registers is specified by using maximum and minimum values separated by a colon (e.g., [4:2]). Bits within an SRAM push/pop register can be specified by <msb:lsb>. If [addr] is omitted, the state refers to the entire array of push/pop registers.

**Example 1:**
```
> dep sc.push_pop_regs[0] = 0xff
```
Deposit a value into push/pop register 0.

**Example 2:**
```
> ex sc.push_pop_regs[0]
```
Examine the result of Example 1. Example display:
```
array:sc.push_pop_regs[0]<22:0>   0000FF
```

## A.2.14 SDRAM

Used with the deposit and examine commands. Refers to the SDRAM array of quadwords, the size of which is set when you initialize memory with the mem_init Transactor command.

**Format:** `{chipname}.sdram{[addr]<msb:lsb>}`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`sdram{[addr]<msb:lsb>}`**
[addr] specifies a quadword offset into SDRAM. Valid [addr] values are 0 to the amount in bytes specified in the mem_init Transactor command divided by 8 (since there are 8 bytes in one quadword). Single locations are specified by a single value (e.g., [0]). A range of locations is specified by using maximum and minimum values separated by a colon (e.g., [1023:0]). Bits within an SDRAM quadword can be specified by <msb:lsb>. Deposits can only be performed for values of 32 bits or less. Examines can occur on 0 to 64 bit values. If [addr] is omitted, the state refers to SDRAM as a whole.

**Example 1:**
```
> chip
AA-1200 chip, "", was instantiated
> mem_init sdram 1m
> dep sdram[0] = 0xffffffff
memory:sdram[0]<31:0>   FFFFFFFF
```
Instantiate a chip, use the mem_init command to set the size of SDRAM, and deposit a value. Then examine the result. The lines preceded by a prompt character (>) are the ones you type in.

**Example 2:**
```
> ex sdram[0]
memory:sdram[0]<31:0>   FFFFFFFF
```
Examine the result of Example 1:

# A.3 Simulation States

## A.3.1 Specify Microengine(s) for GOTO

Used with the deposit and examine commands. A 6-bit value that specifies the Microengine(s) to which a GOTO command applies when used with a label or control store address. When GOTO executes, the Transactor runs until the specified Microengine(s) reach the label or address and ignores any Microengines that are not enabled.

**Format:** `{chipname}.art.fbox_mask`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`art`** Constant that specifies this is a model artifact.

**`fbox_mask`** A six-bit mask that specifies which Microengine(s) should be enabled. The LSB signifies Microengine 0 and the MSB signifies Microengine 5.

**Example 1:**
```
> dep art.fbox_mask = 0x3f
```
Enable all Microengines.

**Example 2:**
```
> dep art.fbox_mask = 0x4
```
Enable only Microengine 2.

## A.3.2 Specify Thread(s) for GOTO

Used with the deposit and examine commands. A 4-bit value that specifies the thread(s) to which a GOTO command applies when used with a label or control store address. When GOTO executes, the Transactor runs until the specified thread(s) reach the label or address and ignores any threads that are not enabled. There are two formats for this state, either of which can be specified independently of the other. The first allows you to specify a Microengine in a context and the second one applies to all Microengines in a context.

**Format:**
```
{chipname}.fx.art.ctx_mask
{chipname}.art.ctx_mask
```

**chipname** — An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** — The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**art** — Constant that specifies this is a model artifact.

**ctx_mask** — A four-bit mask that specifies which threads should be enabled. The LSB signifies thread 0 and the MSB signifies thread 3.

**Example 1:**
```
> dep f0.art.ctx_mask = 0xf
```
Enable all threads in Microengine 0.

**Example 2:**
```
> dep f0.art.ctx_mask = 0x8
```
Enable only thread 2 in Microengine 0.

**Example 3:**
```
> dep art.ctx_mask = 0x8
```
Enable thread 2 for all Microengines.

## A.3.3 Report Core Clock Cycles

Used with the examine command. Indicates the number of StrongARM Core clock cycles simulated during the current session.

**Format:**
```
sim.core_clk_cycle
```

**sim** — Constant that specifies this is a simulator state.

**core_clk_cycle** — Specifies that StrongARM Core clock cycles should be displayed.

**Example:**
```
0> ex sim.core_clk_cycle
sim.core_clk_cycle<31:0>: 0x00000000  (0)  (model variable)
```
Show how many StrongARM Core clock cycles have been simulated during the current session.

## A.3.4 Core Clock Frequency

Used with the deposit and examine commands. Specifies the StrongARM Core clock frequency in MHz.

**Format:** **sim**.core_clk_freq

**sim** Constant that specifies this is a simulator state.

**core_clk_freq** Specifies that the StrongARM Core clock frequency is set or displayed.

**Example:**
```
0> ex sim.core_clk_freq
sim.core_clk_freq<31:0>: 0x000000A6  (166)  (model variable)
```
Show the StrongARM Core clock frequency in MHz.

## A.3.5 Core Clock Period

Used with the deposit and examine commands. Specifies the number of simulation time units per StrongARM Core clock cycle.

**Format:** **sim**.core_clk_period

**sim** Constant that specifies this is a simulator state.

**core_clk_period**
Specifies that the StrongARM Core clock period is set or displayed.

**Example:**
```
0> ex sim.core_clk_period
sim.core_clk_period<31:0>: x00000006  (6)  (model variable)
```
Show the number of simulation time units per StrongARM Core clock cycle.

## A.3.6 Report FBus Clock Cycles

Used with the examine command. Indicates the number of FBus clock cycles simulated during the current session.

**Format:** **sim**.fbus_clk_cycle

**sim** Constant that specifies this is a simulator state.

**fbus_clk_cycle**
Specifies that FBus clock cycles should be displayed.

**Example:**
```
0> ex sim.fbus_clk_cycle
sim.fbus_clk_cycle<31:0>: 0x00000000  (0)  (model variable)
```
Show how many FBus clock cycles have been simulated during the current session.

## A.3.7 FBus Clock Frequency

Used with the deposit and examine commands. Specifies the FBus clock frequency in MHz.

**Format:** **sim**.fbus_clk_freq

**sim** Constant that specifies this is a simulator state.

**fbus_clk_freq** Specifies that the FBus clock frequency is set or displayed.

**Example:**
```
0> ex sim.fbus_clk_freq
sim.fbus_clk_freq<31:0>: 0x00000058  (88)  (model variable)
```
Show the FBus clock frequency in MHz.

## A.3.8 FBus Clock Period

Used with the deposit and examine commands. Specifies the number of simulation time units per FBus clock cycle.

**Format:** **sim**.fbus_clk_period

**sim** Constant that specifies this is a simulator state.

**fbus_clk_period** Specifies that the FBus clock period is set or displayed.

**Example:**
```
0> ex sim.fbus_clk_period
sim.fbus_clk_period<31:0>: 0x0000000C  (12)  (model variable)
```
Show the number of simulation time units per FBus clock cycle.

## A.3.9 Report PCI Clock Cycles

Used with the examine command. Indicates the number of PCI clock cycles simulated during the current session.

**Format:** **sim**.pci_clk_cycle

**sim** Constant that specifies this is a simulator state.

**pci_clk_cycle** Specifies that PCI clock cycles should be displayed.

**Example:**
```
0> ex sim.pci_clk_cycle
sim.pci_clk_cycle<31:0>: 0x00000000  (0)  (model variable)
```
Show how many PCI clock cycles have been simulated during the current session.

## A.3.10 PCI Clock Frequency

Used with the deposit and examine commands. Specifies the PCI clock frequency in MHz.

**Format:** **sim**.pci_clk_freq

**sim** Constant that specifies this is a simulator state.

**pci_clk_freq** Specifies that the PCI clock frequency is set or displayed.

**Example:**

```
0> ex sim.pci_clk_freq
sim.pci_clk_freq<31:0>: 0x00000058  (88)  (model variable)
```

Show the PCI clock frequency in MHz.

## A.3.11 PCI Clock Period

Used with the deposit and examine commands. Specifies the number of simulation time units per PCI clock cycle.

**Format:** **sim**.pci_clk_period

**sim** Constant that specifies this is a simulator state.

**pci_clk_period**

Specifies that the PCI clock period is set or displayed.

**Example:**

```
0> ex sim.pci_clk_period
sim.pci_clk_period<31:0>: 0x0000000C  (12)  (model variable)
```

Show the number of simulation time units per PCI clock cycle.

## A.3.12 Execution Order

Used with the **deposit** and **examine** commands. Specifies the order in which watches, callbacks and the foreign_model_post_sim routine is called after each Transactor simulation event is executed.

Because watches, callbacks, and foreign_model_post_sim offer the user three different mechanisms to actively change state and passively observe state, the execution order of these mechanisms is important for correct operation. For example, passive reads (e.g., informational debug statements) before an active state change may cause incorrect debugging data to be displayed. As another example, incorrect behavior could occur if active state changes are not evaluated in the correct order.

Using the notation W for watches, C for callbacks, and P for foreign_model_post_sim, the legal values for execution order are as follows:

| | |
|---|---|
| 0 | W-C-P |
| 1 | W-P-C |
| 2 | C-W-P |
| 3 | C-P-W |
| 4 | P-W-C |
| 5 | P-C-W |

**Format:** **`sim`**`.post_sim_exec_order`

**`sim`** Constant that specifies this is a simulator state.

**`post_sim_exec_order`**
Specifies that the execution order is set or displayed.

**Example:**
```
0> ex sim.post_sim_exec_order
sim.post_sim_exec_order<2:0>: 0x0  (0)  (model variable)
```
Show the execution order of watches, callbacks, and the foreign_model_post_sim routine. In this case, the value displayed is 0, which represents W-C-P.

## A.3.13 Report Error Count

Used with the examine command. Indicates the number of errors flagged during the current simulation session.

**Format:** **`sim`**`.error_count`

**`sim`** Constant that specifies this is a simulator state.

**`error_count`** Specifies that an error count should be displayed.

**Example:**
```
> ex sim.error_count
sim.error_count<31:0>: 0x00000000  (0)  (model variable)
```
Show how many errors occurred during the current simulation session.

## A.3.14 Error Handle Mode

Used with the deposit and examine commands. Sets or displays what action is taken when a simulation error occurs. Valid settings are as follows:

-1 Suppress and ignore the error.

0 Print the error, but ignore its occurrence. Simulation continues and **sim.error_count** is not incmented.

1 Print the error, increment **sim.error_count**, and halt simulation if simulation is in progress.

2 Print the error, increment **sim.error_count**, and halt simulation and/or any command line/script file execution in progress.

3 Print the error, increment **sim.error_count**, halt all execution, and exit the simulator.

The default setting is 2.

**Format:** `sim.error_handle_mode`

`sim` Constant that specifies this is a simulator state.

`error_handle_mode`
Specifies that error handle mode is set or displayed.

**Example 1:**
```
> ex sim.error_handle_mode
sim.error_handle_mode<31:0>: 0x00000002  (2) (model variable)
```
Examine the error handle mode.

**Example 2**:
```
> dep sim.error_handle_mode = 3
sim.error_handle_mode[0]<31:0>:0x00000003(2)(model variable)
```
Set the error handle mode to 3.

## A.3.15 Stop Simulation on P1 or P3

Used with the deposit and examine commands. This state relates to the **goto**, **ubreak**, and **go fbox/ctx** commands. It specifies whether to stop simulation by examining the P1 microword state or the P3 microword state. The P3 state is desired in most cases, because a P1 state stop may be a “false” one if the P1 microword aborts in P2 or P3.

**Format:** `sim.goto_and_ubreak_key_on_p1`

`sim` Constant that specifies this is a simulator state.

`goto_and_ubreak_key_on_p1`
Specifies goto and ubreak key on P1.

**Example:**
```
0> ex sim.goto_and_ubreak_key_on_p1
sim.goto_and_ubreak_key_on_p1<31:0>: 0x00000000 (0)  (model variable)
```

## A.3.16 Halt Simulation

Used with the deposit and examine commands. When set to 1, halts a running simulation at the end of the current cycle. Typically used with a watch statement when you want the simulation to stop when a specific set of conditions exists. Set to 0 by default.

**Format:** `sim.halt`

**`sim`** Constant that specifies this is a simulator state.

**`halt`** Specifies a halt.

**Example 1:**
```
> ex sim.halt
sim.halt<0>:          0x0   (0)   (model variable)
```
Examine the halt state. It is 0 when the simulation is running.

**Example 2:**
```
> dep sim.halt = 1
sim.halt[0]<0>: 0x1   (1)   (model variable)
Model prematurely stopped by user action.
```
Halt the simulation.

## A.3.17 Suppress Comments

Used with the deposit and examine commands. When set to 1, suppresses the display of lines that contain only comments. Lines that contain instructions and comments are displayed. Set to 0 by default.

**Format:** `sim.prune_prefix_comments`

**`sim`** Constant that specifies this is a simulator state.

**`prune_prefix_comments`**
Specifies the supression of prefix comments.

**Example 1:**
```
> ex sim.prune_prefix_comments
sim.prune_prefix_comments<0>: 0x0   (0)   (model variable)
```
Examine the suppress comments state. It is 0 by default.

**Example 2:**
```
> dep sim.prune_prefix_comments = 1
sim.prune_prefix_comments[0]<0>: 0x1   (1)   (model variable)
```
Suppress comments.

## A.3.18 Suppress Debug Information

Used with the deposit and examine commands. When set to 1, suppresses the display of all debugging information. Set to 0 by default.

**Format:** **sim**.silent

**sim** Constant that specifies this is a simulator state.

**silent** Specifies the supression of debug information.

**Example 1:**
```
> ex sim.silent
sim.silent<0>:          0x0  (0)  (model variable)
```
Examine the suppress debug information state. It is 0 by default.

**Example 2:**
```
> dep sim.silent = 1
sim.silent[0]<0>:       0x1  (1)  (model variable)
```
Suppress debug information.

## A.3.19 Report Simulation Time

Used with the examine command. Indicates the current simulation time.

**Format:** **sim**.time

**sim** Constant that specifies this is a simulator state.

**time** Specifies simulation time.

**Example:**
```
0> ex sim.time
sim.time<31:0>: 0x00000000  (0)  (model variable)
```
Show the surrent simulation time.

## A.3.20 Specify Debug Verbosity Level

Used with the deposit and examine commands. For Microengines, this state determines the level of verbosity of information that is displayed when you execute the debug command. A value of 0 (disable information display) through 4 (display information with highest verbosity level) is valid. For the FBI, SRAM, and SDRAM, only values of 0 (disable information display) or 1 (enable information display) are used.

**Format:** `{chipname}.unit.art.debug`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`unit`** Specifies the unit. Valid names are:

- fx A Microengine, where x specifies the Microengine number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.
- fbi FBI.
- dc SRAM.
- sc SDRAM.

**`art`** Constant that specifies this is a model artifact state.

**`debug`** Specifies debug verbosity.

**Example 1:**

```
> dep f0.art.debug_p0 = 1//view stage 0
> dep f0.art.debug = 1
> debug
====P0 fbox stage (ustore lookup) for f0. in ctx:
0 at fbox cycle 5====
ustore addr = 4 ==> A00A3038
sram_bi[read, $val, val_ptr, 0, word_cnt_1],
defer[1], ctx_swap      ;$val = val
```

View only execution stage 0 (refer to the Specify Reported Execution Stages state for more information on execution stages).

**Example 2:**

```
> dep f0.art.debug_p4 = 1 // view only pipeline stage 4
f0.art.debug_p4[0]<0>: 0x1  (1)  (model variable)


> debug
ctx_0:  saved_pc:  6;  signals--> SELF ;  wake_enable--> SELF
ctx_1:  saved_pc:  1;  signals--> SELF ;  wake_enable-->
ctx_2:  saved_pc:  0;  signals--> SELF ;  wake_enable--> SELF
ctx_3:  saved_pc:  0;  signals--> SELF ;  wake_enable--> SELF
ping ref (output):  READ; addr = 00000001; fbox = 0 ctx = 0
burst_cnt = 1  (issued at fbox cycle = 7; uPC = 3)
pong ref (input ):  READ; addr = 00000002; fbox = 0 ctx = 0
burst_cnt = 1  (issued at fbox cycle = 8; uPC = 4)
====P4 fbox stage (write result) for f0. in ctx: 0 at fbox
cycle 9====
uword addr = 4
sram_bi[read, $val, val_ptr, 0, word_cnt_1],
defer[1], ctx_swap      ;$val = val

W_BUS:  00000002
```

View only execution stage 4 (refer to the Specify Reported Execution Stages state for more information on execution stages).

## A.3.21 Specify Reported Execution Stages

Used with the deposit and examine commands. Determines which micropipeline execution stages are displayed when you execute the debug command. When the fx.art.debug state is enabled to display execution pipeline stage information, this state disables the display of the specified Microengine execution pipeline stage information. For example, when f0.art.debug is set to 4, the Microengine execution pipeline stage information for each stage will be displayed. If f0.art.debug_p3 is set to 0 then only stages 0,1,2, and 4 will be displayed. Depositing a value of 0 disables the display.

**Format:** `{chipname}.fx.art.debug_px`

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x specifies the number. Valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0). You can use the wildcard notation f* with the deposit/multi instruction to specify all Microengines.

**art** Constant that specifies this is a model artifact state.

**debug_px** Specifies one of the five execution stages. Valid names are:

- `debug_p0` Execution stage 0.
- `debug_p1` Execution stage 1.
- `debug_p2` Execution stage 2.
- `debug_p3` Execution stage 3.
- `debug_p4` Execution stage 4.

**Example:** Display all execution stages when the debug command is executed. Assume a low level of verbosity (refer to the Specify Debug Verbosity Level state for more information on the verbosity level).

```
> dep f0.art.debug_p0 = 1//view stage 0
> dep f0.art.debug_p1 = 1//view stage 1
> dep f0.art.debug_p2 = 1//view stage 2
> dep f0.art.debug_p3 = 1//view stage 3
> dep f0.art.debug_p4 = 1//view stage 4
> debug
====P4 fbox stage (write result) for f0. in ctx: 0 at fbox cycle 5====
uword addr = 0
.operand_synonym  no_of_links 0x3
start#:
    ld_field_b[cur_ptr, 1111, headptr, 0]   ;cur_head = headptr
W_BUS:  00000001
operand write:  GPR_A[0]=00000001
====P3 fbox stage (ALU/shifter) for f0. in ctx: 0 at fbox cycle 5====
uword addr = 1
    immed[loop_no, no_of_links]     ;loop = the number of links
A operand: 00000000; pre-shift B operand: 00000003; B operand:
00000003
shifter op: left shift 0; ALU op: ALU_OP_B; ALU result: 00000003;
old_ccs: !=0:>=0:no_carry
====P2 fbox stage (operand lookup) for f0. in ctx: 0 at fbox cycle 5===
uword addr = 2
```

```
loop_add#:
    alu_shf_ri[val_ptr,cur_ptr, +, 1, 0] ;val_ptr = cur_ptr +1
A operand:  00000001  (selected from writeback bypassed data);
B operand:  00000001  (selected from immed data);
====P1 fbox stage (initial decode) for f0. in ctx: 0 at fbox cycle 5===
uword addr = 3
    sram_bi[read, $next, cur_ptr, 0, word_cnt_1] ;$next = next
====P0 fbox stage (ustore lookup) for f0. in ctx: 0 at fbox cycle 5===
ustore addr = 4 ==> A00A3038
    sram_bi[read, $val, val_ptr, 0, word_cnt_1],
defer[1], ctx_swap      ;$val = valdep art.ctx_mask = 0x8
```

## A.3.22 Chip Version

Used with the deposit and examine commands. These states specify the chip version for all chips to be instantiated.

**Format:**

```
param.major_stepping
param.minor_stepping
```

| | |
|---|---|
| **`param`** | Constant that specifies this is a simulation parameter state. |
| **`major_stepping`** | Specifies the major stepping where 0, 1, 2,... represent steppings A, B, C,… |
| **`minor_stepping`** | Specifies the minor stepping, e.g., 0, 1, …. |

***Note:*** A deposit to these states should only be done prior to instantiating any chips. Changing these values after a chip has been instantiated will have unpredictable results.

**Example:**

```
fbox:0> dep param.major_stepping = 2
param.major_stepping[0]<31:0>: 0x00000002  (2)  (model variable)

fbox:0> dep param.minor_stepping = 0
param.minor_stepping[0]<31:0>: 0x00000000  (0)  (model variable)
```

Set stepping to C0.

## A.3.23 Suppressing Outstanding Signal Errors or Warnings

Used with the **deposit** and **examine** commands. These states control the reporting of outstanding signal errors or warnings.

**Format:**

```
param.suppress_outstanding_sig_err
param.suppress_outstanding_sig_warn
```

**`param`** Constant that specifies this is a simulation parameter state.

**`suppress_outstanding_sig_err`** Specifies that the error reported when a signal is received when the same signal is already outstanding should be suppressed. A warning will be reported unless it too is suppressed

**`suppress_outstanding_sig_warning`** Specifies that the warning reported when a signal is received when the same signal is already outstanding should be suppressed. This state has no effect if the error is not suppressed.

*Note:* Use this option with caution, since in most cases, the error indicates a programming error

**Example:**

```
fbox:0> dep param.suppress_outstanding_sig_err = 1
param.suppress_outstanding_sig_err[0]<0>:  0x1  (1)  (model variable)

fbox:0> dep param.suppress_outstanding_sig_warn = 1
param.suppress_outstanding_sig_warn[0]<0>:  0x1  (1)  (model variable)
```

Suppress both errors and warnings.

# A.4 Statistics States

## A.4.1 Compute Cycles

Used with the examine command. Displays the efficiency of Microengine compute cycles. There are three types of cycles: non-aborted (work is performed), aborted (no work is performed but the Microengine is still executing), and stalled (no work is performed because the Microengine stalls).

**Format:** `{chipname}.fx.stat.compute_cycles`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fx`** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**`stat`** Constant that specifies this is a model statistic.

**`compute_cycles`**
Specifies that the statistic type is compute cycles.

**Example:**

```
> ex f0.stat.compute_cycles
f0.stat.compute_cycles:statistics collector:
"Breakdown of compute cycles for f0"
samples = 24;  min = 0;  ave = 4;  max = 9;  data units = cycles

ctx 0 non-aborted cycles:  6     (25%) ************
ctx 0 aborted cycles:      1     (4%)  **
ctx 0 stalled cycles:      0     (0%)
ctx 1 non-aborted cycles:  6     (25%) ************
ctx 1 aborted cycles:      0     (0%)
ctx 1 stalled cycles:      0     (0%)
ctx 2 non-aborted cycles:  6     (25%) ************
ctx 2 aborted cycles:      0     (0%)
ctx 2 stalled cycles:      0     (0%)
ctx 3 non-aborted cycles:  5     (21%) **********
ctx 3 aborted cycles:      0     (0%)
ctx 3 stalled cycles:      0     (0%)
no ctx active cycles:      0     (0%)
```

## A.4.2 SDRAM Read Latency

Used with the examine command. Displays the latency distribution for SDRAM references that return data to a Microengine.

**Format:** `{chipname}.fx.stat.sdram_read_latency`

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**stat** Constant that specifies this is a model statistic.

**sdram_read_latency**
Specifies that the statistic type is sdram_read_latency.

**Example:**

```
> ex f0.stat.sdram_read_latency
f0.stat.sdram_read_latency:statistics collector:
```

## A.4.3 SRAM Read Latency

Used with the examine command. Displays the latency distribution for SRAM references that return data to a Microengine.

**Format:** `{chipname}.fx.stat.sram_read_latency`

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**stat** Constant that specifies this is a model statistic.

**sram_read_latency**
Specifies that the statistic type is sram_read_latency.

**Example:**

```
> ex f0.stat.sram_read_latency
f0.stat.sram_read_latency:statistics collector:  "latency
distribution for SRAM non-read_lock refs returning data to
f0"
    samples = 10;  min = 17;  ave = 18;  max = 24;  data units
= cycles
    sample distrib |  4|  5|   1|
    percent        |40%|50%| 10%|
    cumul percent  |40%|90%|100%|
    cycles          | 17| 18|  24|
```

## A.4.4 SRAM Read Lock Latency

Used with the **examine** command. Displays the latency distribution for SRAM read lock references that return data to a Microengine.

**Format:** `{chipname}.fx.stat.sram_read_lk_latency`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fx`** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**`stat`** Constant that specifies this is a model statistic.

**`sram_read_lk_latency`**
Specifies that the statistic type is sram_read_lk_latency.

**Example:**
```
> ex f0.stat.sram_read_lk_latency
```

## A.4.5 Thread PC During Idle Cycle

Used with the examine command. For a given Microengine, displays a thread's microstore address when no threads are actively running.

**Format:** `{chipname}.fx.stat.threadx_pc_during_idle_cycles`

**`chipname`** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**`fx`** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**`stat`** Constant that specifies this is a model statistic.

**`threadx_pc_during_idle_cycles`**
Specifies the thread. Valid names are:

- `thread0_pc_during_idle_cycles` Thread 0.
- `thread1_pc_during_idle_cycles` Thread 1.
- `thread2_pc_during_idle_cycles` Thread 2.
- `thread3_pc_during_idle_cycles` Thread 3.

**Example:**
```
> ex f0.stat.thread0_pc_during_idle_cycles
```

## A.4.6 Thread Latency

Used with the examine command. Displays information on thread latency, defined as the time between the occurrences of the following two states: {chipname}.fx.art.thread_start_addr and {chipname}.fx.art.thread_stop_addr.

**Format:** `{chipname}.fx.stat.thread_latency`

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**stat** Constant that specifies this is a model statistic.

**thread_latency**
Specifies thread latency.

**Example:**
```
> ex f0.stat.thread_latency
```

## A.4.7 Thread Latency Start and Stop Addresses

Used with the deposit and examine commands. These two states define the beginning and end points of an interval in which the thread latency is measured (see the state definition for fx.stat.thread_latency).

**Format:**
```
{chipname}.fx.art.thread_latency_start_addr
{chipname}.fx.art.thread_latency_stop_addr
```

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**art** Constant that specifies this is a model artifact state.

**thread_latency_start_addr**
An address in the specified Microengine control store that indicates the starting point for measuring the thread latency.

**thread_latency_stop_addr**
An address in the specified Microengine control store that indicates the end point for measuring the thread latency.

**Example:**
```
> f0.art.thread_start_addr = uaddr(f0.ctl.ustore, "label1#");
> f0.art.thread_stop_addr = uaddr(f0.ctl.ustore, "label2#");
> goto /s label2#
> ex f0.stat.thread_latency
```
Set a beginning point at label1# and an end point at label2#. Go to the end point and display the thread latency.

## A.4.8 Time Slice

Used with the examine command. Displays time slice information for a specified thread. A time slice starts when a thread starts and ends when the thread is put to sleep.

**Format:** `{chipname}.fx.stat.time_slice_threadx`

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**stat** Constant that specifies this is a model statistic.

**time_slice_threadx**
Specifies the time slice thread. Valid names are:

- `time_slice_thread0` Thread 0.
- `time_slice_thread1` Thread 1.
- `time_slice_thread2` Thread 2.
- `time_slice_thread3` Thread 3.

**Example:**
```
> ex f0.stat.time_slice_thread0
```

## A.4.9 Time Slice Thread Idle

Used with the examine command. Displays the amount of time during which a specified thread is idle during a time slice. A time slice starts when a thread starts and ends when the thread is put to sleep.

**Format:** `{chipname}.fx.stat.time_slice_threadx_idle`

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**fx** The Microengine number, where x indicates the number and valid Microengine numbers are 0 through 5 (e.g., f0 for Microengine 0).

**stat** Constant that specifies this is a model statistic.

**time_slice_threadx_idle**
Specifies the time slice idle thread. Valid names are:

- `time_slice_thread0_idle` Thread 0.
- `time_slice_thread1_idle` Thread 1.
- `time_slice_thread2_idle` Thread 2.
- `time_slice_thread3_idle` Thread 3.

**Example:**
```
> ex f0.stat.time_slice_thread0_idle
```

## A.4.10 SDRAM Bandwidth

Used with the examine command. Displays bandwidth distribution statistics for SDRAM.

**Format:** {chipname}.**dc.stat**.bandwidth

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**dc** Constant that specifies this state refers to SDRAM.

**stat** Constant that specifies this is a model statistic.

**bandwidth** Specifies that this statistic relates to bandwidth data.

**Example:**

```
> ex dc.stat.bandwidth
dc.stat.bandwidth:statistics collector:  "SDRAM Bandwidth
Distribution for dc"
samples = 20;  min = 0;  ave = 1;  max = 3;  data units =
sdram cycles
read cycles:            8     (40%) ************
write cycles:           0     (0%)
rd->wr dead cycles:     0     (0%)
idle cycles:           12      (60%)   ********************
```

## A.4.11 SDRAM Even Bank Queue Fullness

Used with the examine command. Displays fullness statistics for the SDRAM even bank queue.

**Format:** {chipname}.**dc.stat**.even_bank_queue_fullness

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**dc** Constant that specifies this state refers to SDRAM.

**stat** Constant that specifies this is a model statistic.

**even_bank_queue_fullness**
Specifies that this statistic relates to even bank queue fullness.

**Example:**

```
> ex dc.stat.even_bank_queue_fullness
dc.stat.even_bank_queue_fullness:statistics collector:
"SDRAM Even Bank Queue Fullness Statistics for dc; queue size
= 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.12 SDRAM Odd Bank Queue Fullness

Used with the examine command. Displays fullness statistics for the SDRAM odd bank queue.

**Format:** {chipname}.**dc.stat**.odd_bank_queue_fullness

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**dc** Constant that specifies this state refers to SDRAM.

**stat** Constant that specifies this is a model statistic.

**odd_bank_queue_fullness**
Specifies that this statistic relates to odd bank queue fullness.

**Example:**

```
> ex dc.stat.odd_bank_queue_fullness
dc.stat.odd_bank_queue_fullness:statistics collector:
"SDRAM Odd Bank Queue Fullness Statistics for dc; queue size
= 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.13 SDRAM Order Queue Fullness

Used with the examine command. Displays fullness statistics for the SDRAM order queue.

**Format:** {chipname}.**dc.stat**.order_queue_fullness

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**dc** Constant that specifies this state refers to SDRAM.

**stat** Constant that specifies this is a model statistic.

**order_queue_fullness**
Specifies that this statistic relates to order queue fullness.

**Example:**

```
> ex dc.stat.order_queue_fullness
dc.stat.order_queue_fullness:statistics collector:  "SDRAM
Order Queue Fullness Statistics for dc; queue size = 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.14 SDRAM Priority Queue Fullness

Used with the examine command. Displays fullness statistics for the SDRAM priority queue.

**Format:** {chipname}.**dc.stat**.priority_queue_fullness

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**dc** Constant that specifies this state refers to SDRAM.

**stat** Constant that specifies this is a model statistic.

**priority_queue_fullness**
Specifies that this statistic relates to priority queue fullness.

**Example:**

```
> ex dc.stat.priority_queue_fullness
dc.stat.priority_queue_fullness:statistics collector:
"SDRAM Priority Queue Fullness Statistics for dc; queue size
= 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.15 SRAM Bandwidth

Used with the examine command. Displays bandwidth distribution statistics for SRAM.

**Format:** {chipname}.**sc.stat**.bandwidth

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**sc** Constant that specifies this state refers to SRAM.

**stat** Constant that specifies this is a model statistic.

**bandwidth** Specifies that this statistic relates to bandwidth data.

**Example:**

```
> ex sc.stat.bandwidth
sc.stat.bandwidth:statistics collector:  "SRAM Bandwidth
Distribution for sc"
samples = 20;  min = 0;  ave = 1;  max = 3;  data units = sram
cycles
read cycles:            8     (40%) **************
write cycles:           0     (0%)
rd->wr dead cycles:     0     (0%)
idle cycles:            12     (60%) ************************
```

## A.4.16 SRAM Order Queue Fullness

Used with the examine command. Displays fullness statistics for the SRAM order queue.

**Format:** {chipname}.**sc.stat**.order_queue_fullness

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**sc** Constant that specifies this state refers to SRAM.

**stat** Constant that specifies this is a model statistic.

**order_queue_fullness**
Specifies that this statistic relates to order queue fullness.

**Example:**

```
> ex sc.stat.order_queue_fullness
sc.stat.order_queue_fullness:statistics collector:  "SRAM
Order Queue Fullness Statistics for sc; queue size = 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.17 SRAM Priority Queue Fullness

Used with the examine command. Displays fullness statistics for the SRAM priority queue.

**Format:** {chipname}.**sc.stat**.priority_queue_fullness

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**sc** Constant that specifies this state refers to SRAM.

**stat** Constant that specifies this is a model statistic.

**priority_queue_fullness**
Specifies that this statistic relates to priority queue fullness.

**Example:**

```
> ex sc.stat.priority_queue_fullness
sc.stat.priority_queue_fullness:statistics collector:  "SRAM
Priority Queue Fullness Statistics for sc; queue size = 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.18 SRAM Read Lock Queue Fullness

Used with the examine command. Displays fullness statistics for the SRAM read lock queue.

**Format:** {chipname}.**sc.stat**.rd_lk_queue_fullness

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**sc** Constant that specifies this state refers to SRAM.

**stat** Constant that specifies this is a model statistic.

**rd_lk_queue_fullness**
Specifies that this statistic relates to read lock queue fullness.

**Example:**

```
> ex sc.stat.rd_lk_queue_fullness
sc.stat.rd_lk_queue_fullness:statistics collector:  "SRAM
Read Lock Queue Fullness Statistics for sc; queue size = 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.19 SRAM Read Queue Fullness

Used with the examine command. Displays fullness statistics for the SRAM read queue.

**Format:** {chipname}.**sc.stat**.read_queue_fullness

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**sc** Constant that specifies this state refers to SRAM.

**stat** Constant that specifies this is a model statistic.

**read_queue_fullness**
Specifies that this statistic relates to read queue fullness.

**Example:**

```
> ex sc.stat.read_queue_fullness
sc.stat.read_queue_fullness:statistics collector:  "SRAM
Read Queue Fullness Statistics for sc; queue size = 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.20 SRAM State Machine Distribution

Used with the **examine** command. Displays state machine distribution statistics for SRAM.

**Format:** {chipname}.**sc.stat**.state_machine_dist

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**sc** Constant that specifies this state refers to SRAM.

**stat** Constant that specifies this is a model statistic.

**state_machine_dist**
Specifies that this statistic relates to state machine distribution.

**Example:**

```
> ex sc.stat.state_machine_dist
sc.stat.state_machine_dist:statistics collector:  "SRAM
State Machine Distribution Statistics for sc; queue size = 16"
samples = 20;  min = 0;  ave = 0;  max = 0;  data units =
queue entries
sample distrib |  20|
percent        |100%|
cumul percent  |100%|
queue entries |   0|
```

## A.4.21 Command Bus Bandwidth

Used with the examine command. Displays bandwidth distribution statistics for the command bus.

**Format:** {chipname}.**sc.stat**.cmd_bus_bandwidth

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**sc** Constant that specifies this state refers to SRAM.

**stat** Constant that specifies this is a model statistic.

**cmd_bus_bandwidth**
Specifies that this statistic relates to the command bus bandwidth.

**Example:**

```
> ex sc.stat.cmd_bus_bandwidth
stat.cmd_bus_bandwidth:statistics collector:  "Reference
Packet Bandwidth Distribution for chip "" "
samples = 60;  min = 0;  ave = 0;  max = 2;  data units = fbox
cycles
no refs:            45   (75%) ****************************
SDRAM refs:          0    (0%)
SRAM refs:          15   (25%) ************
SRAM_CSR refs:       0    (0%)
PCI refs:            0    (0%)
FBI refs:            0    (0%)
Scratch-Pad refs:     0     (0%)
```

## A.4.22 SBUS Pull Bus Bandwidth

Used with the examine command. Displays bandwidth distribution statistics for the SBUS pull bus.

**Format:** {chipname}**.stat**.sbus_pull_bandwidth

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**stat** Constant that specifies this is a model statistic.

**sbus_pull_bandwidth**
Specifies that this statistic relates to the SBUS pull bus bandwidth.

**Example:**

```
> ex stat.sbus_pull_bandwidth
stat.sbus_pull_bandwidth:statistics collector:  "SRAM data
pull bus bandwidth distribution for chip "" "
   samples = 60;  min = 15;  ave = 23;  max = 31;  data units
= fbox cycles
SRAM <- Fbox 0 data:       0     (0%)
SRAM <- Fbox 1 data:       0     (0%)
SRAM <- Fbox 2 data:       0     (0%)
SRAM <- Fbox 3 data:       0     (0%)
SRAM <- Fbox 4 data:       0     (0%)
SRAM <- Fbox 5 data:       0     (0%)
SRAM <- FBI data:          0     (0%)
SRAM <- Fbox 0 CSR data:   0     (0%)
SRAM <- Fbox 1 CSR data:   0     (0%)
SRAM <- Fbox 2 CSR data:   0     (0%)
SRAM <- Fbox 3 CSR data:   0     (0%)
SRAM <- Fbox 4 CSR data:   0     (0%)
SRAM <- Fbox 5 CSR data:   0     (0%)
SRAM <- no xfer:          30    (50%) **********************
FBI <- Fbox 0 data:        0     (0%)
FBI <- Fbox 1 data:        0     (0%)
FBI <- Fbox 2 data:        0     (0%)
FBI <- Fbox 3 data:        0     (0%)
FBI <- Fbox 4 data:        0     (0%)
FBI <- Fbox 5 data:        0     (0%)
FBI <- FBI data:           0     (0%)
FBI <- Fbox 0 CSR data:    0     (0%)
FBI <- Fbox 1 CSR data:    0     (0%)
FBI <- Fbox 2 CSR data:    0     (0%)
FBI <- Fbox 3 CSR data:    0     (0%)
FBI <- Fbox 4 CSR data:    0     (0%)
FBI <- Fbox 5 CSR data:    0     (0%)
FBI <- no xfer:           30    (50%) **********************
```

## A.4.23 SBUS Push Bus Bandwidth

Used with the examine command. Displays bandwidth distribution statistics for the SBUS push bus.

**Format:** {chipname}**.stat**.sbus_push_bandwidth

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**stat** Constant that specifies this is a model statistic.

**sbus_push_bandwidth**
Specifies that this statistic relates to the SBUS push bus bandwidth.

**Example:**

```
> ex stat.sbus_push_bandwidth
stat.sbus_push_bandwidth:statistics collector:  "SRAM data
push bus bandwidth distribution for chip "" "
   samples = 60;  min = 15;  ave = 23;  max = 31;  data units
= fbox cycles
SRAM <- Fbox 0 data:       0     (0%)
SRAM <- Fbox 1 data:       0     (0%)
SRAM <- Fbox 2 data:       0     (0%)
SRAM <- Fbox 3 data:       0     (0%)
SRAM <- Fbox 4 data:       0     (0%)
SRAM <- Fbox 5 data:       0     (0%)
SRAM <- FBI data:          0     (0%)
SRAM <- Fbox 0 CSR data:   0     (0%)
SRAM <- Fbox 1 CSR data:   0     (0%)
SRAM <- Fbox 2 CSR data:   0     (0%)
SRAM <- Fbox 3 CSR data:   0     (0%)
SRAM <- Fbox 4 CSR data:   0     (0%)
SRAM <- Fbox 5 CSR data:   0     (0%)
SRAM <- no xfer:          30    (50%) ************************
FBI <- Fbox 0 data:        0     (0%)
FBI <- Fbox 1 data:        0     (0%)
FBI <- Fbox 2 data:        0     (0%)
FBI <- Fbox 3 data:        0     (0%)
FBI <- Fbox 4 data:        0     (0%)
FBI <- Fbox 5 data:        0     (0%)
FBI <- FBI data:           0     (0%)
FBI <- Fbox 0 CSR data:    0     (0%)
FBI <- Fbox 1 CSR data:    0     (0%)
FBI <- Fbox 2 CSR data:    0     (0%)
FBI <- Fbox 3 CSR data:    0     (0%)
FBI <- Fbox 4 CSR data:    0     (0%)
FBI <- Fbox 5 CSR data:    0     (0%)
FBI <- no xfer:           30    (50%) ************************
```

## A.4.24 SDRAM Pull Bus Bandwidth

Used with the examine command. Displays bandwidth distribution statistics for the SDRAM pull bus.

**Format:** {chipname}**.stat**.sdram_pull_bandwidth

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**stat** Constant that specifies this is a model statistic.

**sdram_pull_bandwidth**
Specifies that this statistic relates to the SDRAM pull bus bandwidth.

**Example:**

```
> ex stat.sdram_pull_bandwidth
stat.sdram_pull_data_bandwidth:statistics collector:  "SDRAM
pull data bus bandwidth distribution for chip "" "
   samples = 60;  min = 7;  ave = 7;  max = 7;  data units =
fbox cycles
SDRAM <- Fbox 0 data:    0     (0%)
SDRAM <- Fbox 1 data:    0     (0%)
SDRAM <- Fbox 2 data:    0     (0%)
SDRAM <- Fbox 3 data:    0     (0%)
SDRAM <- Fbox 4 data:    0     (0%)
SDRAM <- Fbox 5 data:    0     (0%)
SDRAM <- R-fifo data:    0     (0%)
SDRAM <- no xfer:       60    (100%)************************
```

## A.4.25 SDRAM Push Bus Bandwidth

Used with the examine command. Displays bandwidth distribution statistics for the SDRAM push bus.

**Format:** {chipname}**.stat**.sdram_push_bandwidth

**chipname** An optional chip name that is meaningful only if the chip has been instantiated with a name.

**stat** Constant that specifies this is a model statistic.

**sdram_pull_bandwidth**
Specifies that this statistic relates to the SDRAM push bus bandwidth.

**Example:**

```
> ex stat.sdram_push_bandwidth
stat.sdram_push_data_bandwidth:statistics collector:  "SDRAM
push data bus bandwidth distribution for chip "" "
   samples = 60;  min = 7;  ave = 7;  max = 7;  data units =
fbox cycles
SDRAM <- Fbox 0 data:    0      (0%)
SDRAM <- Fbox 1 data:    0      (0%)
SDRAM <- Fbox 2 data:    0      (0%)
SDRAM <- Fbox 3 data:    0      (0%)
SDRAM <- Fbox 4 data:    0      (0%)
SDRAM <- Fbox 5 data:    0      (0%)
SDRAM <- R-fifo data:    0      (0%)
SDRAM <- no xfer:       60    (100%)************************
```

# Developer Workbench Shortcuts B

## B.1 Introduction

In the Developer Workbench there are at least three ways to initiate an action:

- A menu command
- A keyboard shortcut
- A toolbar button

The following tables summarize most these tools.

**Table B-3. Developer Workbench Shortcuts—Files**

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | CTRL+N | File, New | Create new file. | Section 2.7.1. |
|  | CTRL+O | File, Open | Open a file. | Section 2.7.2. |
|  | CTRL+S | File, Save | Save a file. | Section 2.7.4. |
| | ALT+F+A | File, Save As | Save copy of file. | Section 2.7.5. |
|  | ALT+F+L | File, Save All | Save all open files. | Section 2.7.6. |
| | ALT+F+U | File, Print Setup | Set up the printer properties. | Section 2.7.8.1. |
| | ALT+F+I | File, Print Preview | View the file to be printed. | Section 2.7.8.2. |
|  | CTRL+P | File, Print | Print file in active window. | Section 2.7.8.3. |
| | ALT+F+F | File, Recent Files | Select from the four most recently opened files. | Section 2.7.2. |
| | ALT+F+C | File, Close, or WIndow, Close | Close the active window. | Section 2.7.3. |

### Table B-4. Developer Workbench Shortcuts—Projects

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
| | ALT+F+W | File, New Project | Create a new project. | Section 2.4.1. |
| | ALT+F+R | File, Open Project | Open a project. | Section 2.4.2. |
| | ALT+F+V | File, Save Project | Save project. | Section 2.4.3 |
| | ALT+F+E | File, Close Project | Close a project. | Section 2.4.4. |
| | ALT+P+I | Project, Insert Assembler Source Files | Insert Assembler source files into a project. | Section 2.7.9.1. |
| | | Project, Insert Compiler Source Files | Insert Compiler source files into a project. | Section 2.7.9.1. |
| | ALT+P+S | Project, Insert Script Files | Insert script files into a project. | Section 2.7.9.1. |
| | ALT+P+U | Project, Update Dependencies | Update project dependencies. | Section 2.8.1. |
| | ALT+P+C | Project, System Configuration | Specify system configuration. | Section 2.5. |

### Table B-5. Developer Workbench Shortcuts—Edit (Sheet 1 of 2)

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | CTRL+Z | Edit, Undo | Undo. | Section 2.7.10. |
|  | CTRL+Y | Edit, Redo | Redo. | Section 2.7.10. |
|  | CTRL+X | Edit, Cut | Cut. | Section 2.7.10. |
|  | CTRL+C | Edit, Copy | Copy selected text. | Section 2.7.10. |
|  | CTRL+V | Edit, Paste | Paste. | Section 2.7.10. |
|  | DELETE | DELETE key | Delete. | Section 2.7.10. |
| | CTRL+A | Edit, Select All | Select all text in the file. | Section 2.7.10. |
|  | CTRL+F | Edit, Find | Find text in a text file. | Section 2.7.10. |

### Table B-5. Developer Workbench Shortcuts—Edit (Sheet 2 of 2)

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | CTRL+SHIFT+F | | Find next. | Section 2.7.10. |
|  | ALT+E+I | Edit, Find in Files | Find in text files. | Section 2.7.12. |
| | CTRL+H | Edit, Replace | In the Replace dialog box, replace the items currently selected in the file with the text in the Replace with box. | Section 2.7.10. |

### Table B-6. Developer Workbench Shortcuts—Bookmarks

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | CTRL+F2 | Edit, Bookmark, Insert/Remove | Insert/Remove bookmark. | Section 2.7.11. |
|  | F2 | Edit, Bookmark, Go To Next | Go to the next bookmark. | Section 2.7.11. |
|  | SHIFT+F2 | Edit, Bookmark, Go To Previous | Go to previous bookmark. | Section 2.7.11. |
|  | CTRL+SHIFT+F2 | Edit, Bookmark, Clear All | Clear all bookmarks. | Section 2.7.11. |

### Table B-7. Developer Workbench Shortcuts—Breakpoints

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | F9 | Debug, Breakpoint, Insert/Remove | Insert/Remove breakpoint. | Section 2.11.10.3, Section 2.11.17.2. |
|  | CTRL+F9 | Debug, Breakpoint, Enable/Disable | Toggle enable/disable. | Section 2.11.10.4. |
|  | ALT+D+B+D | Debug, Breakpoint, Disable All | Disable all breakpoints in active document. | Section 2.11.10.4. |
|  | ALT+D+B+A | Debug, Breakpoint, Enable All | Enable all breakpoints in the active document. | Section 2.11.10.4. |
|  | ALT+D+B+R | Debug, Breakpoint, Remove All | Remove all breakpoints from the active document. | Section 2.11.10.3. |

### Table B-8. Developer Workbench Shortcuts—Builds

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | CTRL+F7 | Build, Assemble | Invoke the Assembler. | Section 2.8.3. |
|  | CTRL+SHIFT+F7 | Build, Compile | Compile. | Section 2.9.3. |
|  | F7 | Build, Build | Link. | Section 2.10. |
|  | ALT+F7 | Build, Rebuild | Rebuild. | Section 2.10. |

### Table B-9. Developer Workbench Shortcuts—Debug

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | F12 | Debug, Start Debugging | Start debugging. | Section 2.11.1. |
|  | CTRL+F12 | Debug, Stop Debugging | Stop debugging. | Section 2.11.1. |
|  | F4 | | Go to next error/tag. | Section 2.7.11, Section 2.7.12, Section 2.8.4, Section 2.9.4. |
|  | SHIFT+F4 | | Go to previous error/tag. | Section 2.7.11, Section 2.8.4, Section 2.9.4. |

### Table B-10. Developer Workbench Shortcuts—Run Control (Sheet 1 of 2)

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | F5 | Debug, Run Control, Go | Start simulation. | Section 2.11.8.10, Section 2.11.9.2. |
|  | SHIFT+F5 | Debug, Run Control, Stop | Stop simulation. | Section 2.11.8.10, Section 2.11.9.2. |
|  | F10 | Debug, Run Control, Step Over | Step over. | Section 2.11.8.3. |
|  | F11 | Debug, Run Control, Step Into | Step into (Compiler thread only). | Section 2.11.8.4. |
|  | SHIFT+F11 | Debug, Run Control, Step Out | Step out (Compiler thread only). | Section 2.11.8.5. |

### Table B-10. Developer Workbench Shortcuts—Run Control (Sheet 2 of 2)

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
|  | CTRL+F10 | Debug, Run Control, Run To Cursor | Run to cursor. | Section 2.11.7.5. |
|  | SHIFT+F10 | Debug, Run Control, Step Microengines | Step Microengines. | Section 2.11.8.2. |
|  | CTRL+SHIFT+F12 | Debug, Run Control, Reset | Reset. | Section 2.11.8.11, Section 2.11.9.4. |

### Table B-11. Developer Workbench Shortcuts—View

| Button | Keyboard | Menu | Action | Reference |
|---|---|---|---|---|
| | CTRL+F6 | Window, <filename> | Make next window active. | Section 2.3.2. |
|  | ALT+V+O | View, Output Window | Toggle visibility of Output window. | Section 2.3.2. |
|  | ALT+V+D+C | View, Debug Window, Command Line | Toggle visibility of Command Line window. | Section 2.11.5. |
|  | ALT+V+D+D | View, Debug Window, Data Watch | Toggle visibility of Data Watch window. | Section 2.11.12. |
|  | ALT+V+D+M | View, Debug Window, Memory Watch | Toggle visibility of Memory Watch window. | Section 2.11.13 |
|  | ALT+V+D+H | View, Debug Window, History | Toggle visibility of History window. | Section 2.11.16. |
|  | ALT+V+D+T | View, Debug Window, Thread Status | Toggle visibility of Thread Status window. | Section 2.11.18. |
|  | ALT+V+D+Q | View, Debug Window, Queue Status | Toggle visibility of Queue Status window. | Section 2.11.17. |
|  | ALT+V+D+I | View, Debug Window, IX Bus Status | Toggle visibility of IX Bus Status window. | Section 2.11.23.4. |
|  | ALT+V+D+R | View, Debug Window, Run Control | Toggle visibility of Run Control window. | Section 2.11.8. |